Melhore seu Código com Domain Driven Design: Focando no que importa

Clube Delphi

Ano 13 :: Edição 150 :: R$ 11,90

DevMedia

AnyDAC

Sua nova opção de acesso a dados

Web com DataSnap no XE3
Saiba o que é possível construir

Padrões de Arquitetura
Em busca da qualidade

LiveBindings e FireMonkey
Ligações com controles DataAware e Objetos

Runtime Designer do Fast Report
Editando relatórios em tempo de execução

ISSN 1517990-7

TENHA ACESSO ILIMITADO A TODO CONTEÚDO DO SITE DEVMEDIA

SEJA UM ASSINANTE MVP!

Tenha acesso garantido a qualquer tema que você possa necessitar!

VIDEOS -AULAS

O assinante MVP acessa todas as vídeo-aulas do site DevMedia. Nenhum outro produto dá essa possibilidade. São mais de **7.000 video-aulas.**

CURSOS

Nosso site possui atualmente **+ de 250 cursos online** com a qualidade que você já conhece e confia. Com o MVP você terá acesso a todos eles.

REVISTAS

Assinante MVP acessa todas as nossas revistas na versão digital, incluindo todas as edições anteriores. Quando pintar aquela dúvida você terá mais de **3.500 artigos para consultar!**

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

A assinatura MVP te dá a oportunidade de aumentar seu conhecimento em diversos assuntos, com qualidade e economia.

Invista na sua carreira se torne um profissional ainda mais completo!

POR APENAS R$59,90 POR MÊS

DevMedia

Acesse: www.devmedia.com.br/mvp

Sumário

Recursos editoriais: Novidades

06 – Arquitetura do AnyDAC
[*Fabrício Hissao Kawata*]

Recursos editoriais: Mentoring/Boas práticas

15 – Melhore seu Código com Domain Driven Design
[*Paulo Quicoli*]

21 – LiveBinding com FireMonkey
[*Giuliano Scombatti Pinto*]

Recursos editoriais: Novidades

30 – Pronto para Web com DataStnap XE3
[*Marcelo Varela*]

Recursos editoriais: Boa prática

36 – Relatórios Dinâmicos com FastReport
[*Wanderley Donizeti Panosso Junior*]

Recursos editoriais: Arquitetura

44 – Padrões de Arquitetura de Aplicações Corporativas
[*Welson Play*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Faça já sua assinatura digital ! | www.devmedia.com.br/es

Faça um *upgrade* em sua carreira

Em um mercado cada vez mais focado em qualidade, ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lançou uma publicação totalmente especializada em Engenharia de Software. Todos os meses você pode encontrar artigos sobre Metodologias Ágeis; Metodologias tradicionais (document driven); ALM (application lifecycle); SOA (aplicações orientadas a serviços); Análise de sistemas; Modelagem; Métricas; Orientação à Objetos; UML; testes e muito mais. **Assine Já!**

DevMedia

Clube Delphi

Ano 13 - 150ª Edição 2013 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral

Paulo Quicoli
pauloquicoli@gmail.com

Equipe Editorial

Guinther Pauli
guintherpauli@gmail.com

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Giuliano Scombatti Pinto
giuliano@sygnux.com.br

Daniel Sobrinho Laporte
daniel.laporte@gmail.com

Jornalista Responsável

Kaline Dolabella - JP24185

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

EDITORIAL

No último dia 5 de Fevereiro a Embacadero anunciou a compra do AnyDAC, da DA-SOFT Technologies. O AnyDAC é conhecido por ser poderoso, fácil de utilizar e ter altíssima performance para aplicações empresariais. Está pronto para acesso nativo a vários banco de dados e a Embarcadero planeja incorporá-lo em sua nova tendência, o ferramentas de desenvolvimento multi-dispositivo, que incluem Delphi, C++ Builder e RAD Studio. Isso irá garantir a geração de um único código fonte para as diversas plataformas existentes: Windows, Mac, iOS e Android. E como estamos ligados na importância disso para a comunidade, o Fabrício apresenta um artigo que mostra como utilizar o AnyDAC, suas características e arquitetura.

O Giuliano aborda as operações básicas do LiveBindings, mostrando como realizar conexão com controles DataWare. Muitos mostram suas novidades em relação à ligação de controles visuais, objetos de negócio, mas como que fica quando ainda temos uma aplicação tradicional? Esse artigo mostra isso.

Ainda falando acerca de "como fazer", o Marcelo faz uma apresentação dos recursos disponíveis no DataSnap XE 3 para o desenvolvimento WEB. Assuntos como autenticação, sessões e filtros são abordados.

Com a integração do gerador de relatório FastReport ao Rad Studio, a comunidade viu a possibilidade de realizar muita coisa, contudo, alguns recursos não estão disponíveis na versão disponibilizada, como o editor de relatórios em tempo de execução. O Wanderley aborda em seu artigo o que é possível alcançar com esse recurso, criando uma solução bastante prática que pode ser incluída em qualquer software comercial: um editor de cheques.

Mas como nem tudo é "sair programando", o Welson aponta alguns padrões de arquitetura voltados para sistemas empresariais, aponta suas vantagens e desvantagens, e eu também apresento o DDD como uma forma de melhorar a qualidade de seu código. Boa leitura.

Paulo Quicoli
pauloquicoli@gmail.com
Editor Geral
quicoli.wordpress.com
twitter.com/pauloquicoli

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia:
www.devmedia.com.br/mvp

DevMedia

Seu **conhecimento vale muito** na DevMedia

Publique artigos e vídeos gratuitamente no site e participe do programa **PayContent.**

Sua participação pode ser premiada: quanto mais você postar, mais chances de ganhar.

Para saber mais acesse:
www.devmedia.com.br/paycontent

DevMedia

A Arquitetura do AnyDAC

Conheça a nova biblioteca de componentes de acesso a dados da Embarcadero

AnyDAC é o nome da nova biblioteca de componentes de acesso a dados adquirida recentemente pela Embarcadero e que deverá acompanhar o produto daqui pra frente em suas futuras versões. Numa definição mais concreta, o AnyDAC é tido como um conjunto de Componentes de Acesso a Dados Universal (Universal Data Access Components) por possuir uma arquitetura robusta, totalmente projetada para suportar os principais e mais variados bancos de dados disponíveis no mercado. Seu uso provê às aplicações acesso direto e nativo para SGBDs como InterBase, Firebird, MS SQL Server, Oracle, SQLite e PostgreSQL, além de servidores DataSnap. Isso se deve à sua poderosa camada de acesso, que torna o AnyDAC ideal para a construção de aplicações que envolvam uma alta carga de dados, bem como as que demandam um alto nível de performance.

Apesar da aquisição por parte da Embarcadero e sua esperada integração ao Delphi, o que vale destacar neste momento é o estágio atual da biblioteca e sua utilização com o Delphi. Historicamente falando, o AnyDAC foi concebido pela empresa DA-Soft Technologies, a partir de uma base de conhecimento e experiência de mais de uma década na escrita de drivers nativos de banco de dados. Em termos de IDE, até o momento o AnyDAC provê suporte, além do próprio Delphi, também para outro produto da Embarcadero, o C++ Builder, além do Free Pascal (**Nota do DevMan 1**).

Principais características do AnyDAC

A fim de caracterizar melhor o AnyDAC, a seguir são mostradas em termos gerais algumas de suas principais características.

Suporte a diversas ferramentas de desenvolvimento

Em seu estágio atual, o AnyDAC provê suporte essencialmente a quatro IDEs diferentes: FreePascal, Lazarus (**Nota do DevMan 2**), C++ Builder e Delphi. Olhando especificamente para este último, praticamente todas são suas versões suportadas, desde a 5 (Delphi 5, 6, 7, 2005, 2007, 2009, 2010), até chegar à família XE, sempre com referência à plataforma Win32/VCL. Para as mais recentes, especificamente XE2 e XE3, adiciona-se a este grupo as plataformas Win64, MacOS, iOS e o FireMonkey.

Suporte a diversos tipos de servidores de banco de dados e DataSnap

Conforme inicialmente citado, o AnyDAC provê conexões nativas, tanto para os principais servidores de bancos de dados

Resumo DevMan

A Arquitetura do AnyDAC :

Até o Delphi XE3, a opção nativa relacionada a framework de acesso a dados mais popular era o dbExpress. Essa afirmativa é justificada pelas suas características, tais como o suporte a diversos SGBDs diferentes e sua arquitetura robusta. No entanto, para as próximas versões do IDE a tendência é que este cenário se altere e o desenvolvedor ganhe mais uma opção de mesmo porte. Isto porque recentemente a Embarcadero anunciou a aquisição de uma nova biblioteca de componentes de acesso a dados, o AnyDAC, da empresa DA-Soft Technologies. Tal como o dbExpress, o AnyDAC se apresenta também como uma solução robusta e com suporte a múltiplos SGBDs. Obviamente suas características não se restringem a isso, sendo assim, a intenção do presente artigo é dar uma visão geral sobre o AnyDAC, mostrando os detalhes de sua arquitetura, componentes e utilização.

Em que situação o tema é útil:

O Delphi proporciona ao desenvolvedor, recursos suficientes para a construção de aplicações dos mais variados tipos. Dentre estas, as aplicações de banco de dados se destacam, sendo, sem sombra de dúvida, a realidade de grande parte da comunidade. Neste sentido, torna-se uma regra a escolha de um framework de acesso a dados, bem como o devido conhecimento sobre ele. De forma nativa, o próprio Delphi já provê algumas opções, como o BDE, dbExpress, InterBase Express e ADO. Mediante a recente aquisição do AnyDAC pela Embarcadero, mais um item "nativo" é acrescido ao leque de opções do desenvolvedor Delphi, no que diz respeito a acesso a dados.

disponíveis no mercado quanto a servidores DataSnap. A seguir é mostrada uma lista completa deste suporte:
- Sybase Advantage Database Server;
- Sybase SQL Anywhere;
- IBM DB2;
- IBM Informix Dynamic Server;
- Firebird;
- InterBase;
- Blackfish SQL Server;
- Microsoft SQL Server;
- Microsoft Access;
- MySQL;
- Oracle;
- PostgreSQL;
- SQLite;
- Unify SQLBase;
- DataSnap Server.

Nota do DevMan 1

O Free Pascal, formalmente conhecido como Free Pascal Compiler (FPC), é um compilador gratuito para a linguagem Pascal/Object Pascal, disponível para diversas arquiteturas e Sistemas Operacionais, tais como Windows, Linux e Mac OS X. Assim como a grande maioria dos compiladores, sua utilização está atrelada a IDEs (Ambientes de Desenvolvimento Integrado). O exemplo mais popular dentre tais IDEs é atualmente o Lazarus, que fornece uma experiência de desenvolvimento muito próxima a do Delphi.

Nota do DevMan 2

Lazarus é um Ambiente de Desenvolvimento Integrado (ou IDE) elaborado para o compilador Free Pascal. Sua intenção e usabilidade são muito próximas ao Delphi, sendo considerado por muitos uma alternativa ao próprio. Dentre suas inúmeras semelhanças, é possível destacar o uso da linguagem Object Pascal e a construção de aplicações gráficas e de banco de dados. Por outro lado, o grande diferencial do Lazarus é o seu desenvolvimento cross-plaftorm, incluindo a produção de aplicativos não só para Windows e Mac OS X, como também para Linux e FreeBSD.

Além das conexões consideradas nativas, o AnyDAC provê suporte também a conexões intermediárias, nestes casos, fontes de dados (data sources) ODBC e dbExpress (DBX).

API unificada e personalizada

Cada SGBD, dentre os diversos existentes atualmente, apresenta suas próprias particularidades, tais como recursos e instruções SQL. Pelo fato de prover suporte a vários destes SGBDs, o AnyDAC apresenta uma forma de abstrair todas essas diferenças através de uma API unificada. Para isso, ele provê uma série de recursos que contribuem para o processo de desenvolvimento, fundamentalmente no que diz respeito à codificação. Dessa forma, o desenvolvedor consegue escrever de uma forma única, sem se preocupar exclusivamente com as nuances de cada SGBD.

Em complemento às praticidades proporcionadas por conta de sua capacidade unificada, tanto de acesso a SGBDs distintos quanto à sua própria API, o AnyDAC contempla também as funcionalidades específicas de cada banco. Sendo assim, sua API é unificada o suficiente para facilitar a utilização dos recursos comuns dos bancos suportados e personalizada o bastante para prover esta mesma facilidade no aproveitamento das características específicas de cada um. Sendo assim, em aplicações Delphi, para cada SGBD o AnyDAC provê um driver nativo diferente, com acesso às suas características peculiares.

A seguir são mostrados exemplos de recursos específicos de alguns dos SGBDs suportados pelo AnyDAC:

Firebird/InterBase
- Cláusula RETURNING;
- Declarações EXECUTE BLOCK;
- Trabalho com eventos (Events);
- Dialeto de scripts específico – IBScript SQL;
- Serviços específicos de banco, tais com Backup, Restore e Validate.

Oracle
- Trabalho com múltiplos "Oracle Home";
- Tipos de dados exclusivos, tais como CLOB, BFILE e INTERVAL;
- Cláusula RETURNING;
- Cursores;
- DBMS_OUTPUT;
- DBMS_ALERT;
- Dialeto de scripts específico - SQL*Plus SQL.

Microsoft SQL Server
- SQL Azure;
- LocalDB;
- Tipos de dados exclusivos, tais como NCHAR, NVARCHAR, VARCHAR(MAX) e UNIQUEIDENTIFIER (GUID);
- Campos auto-incremento (IDENTITY);
- Query Notifications;
- SQL Server Message Broker;
- Dialeto de scripts específico - ISQL SQL.

PostgreSQL
- Cláusula RETURNING;
- Conexões SSL;
- Tipos de dados estruturados, tais como Array e ADT;
- Colunas OID;
- Tipo de dados Serial;
- Notificações.

Nativamente falando, atualmente no Delphi, muitos destes recursos só estão disponíveis mediante a utilização de bibliotecas específicas de determinado SGBD, ou na maioria dos casos, não disponível. Numa exemplificação simples, o acesso a recursos mais específicos do InterBase, tais como Backup,

Restore e Events, é conseguido através da utilização dos componentes IBX (guia InterBase na Tool Palette). No entanto, em casos como este, a flexibilidade da aplicação é afetada, uma vez que os componentes são todos direcionados a uma única tecnologia de banco de dados. A troca desta tecnologia implicaria na troca de todos os componentes utilizados no projeto. Outra solução ainda seria recorrer a outras bibliotecas de componentes de terceiros, também específicas a uma determinada tecnologia de banco de dados. Em vista deste cenário, numa solução simples e abrangente, o AnyDAC provê de forma centralizada o suporte a diferentes SGBDs e aos recursos específicos de cada um.

Engine de acesso a dados de alto de desempenho

A arquitetura da camada de conjuntos de dados (DataSets) do AnyDAC é inspirada e muito similar à arquitetura do ADO .NET (**Nota do DevMan 3**). Sendo assim, a construção de seus componentes de dados, tais como TADQuery, TADStoredProc e TADTable, é toda baseada nesta engine. A fim de manter a harmonia com o contexto Delphi, o AnyDAC compatibiliza seus componentes DataSet com o comportamento padrão de um TDataSet clássico. Em termos de recursos e características, a seguir são sumarizados os principais, comuns a todos os DataSets da biblioteca:

- Compatibilidade com a engine clássica de conectividade com banco de dados do Delphi, o BDE (Borland Database Engine);
- Compatibilidade com TClientDataSet;
- Rapidez (até 500% mais rápido quem um TClientDataSet);
- Filtros sobre campos calculados (Calculated Fields) e agregados (Aggregate Fields);
- Extensa gama de tipos de dados que são suportados;
- Diversas ações sobre os dados dos DataSets, tais como Copy, Merge e Assign;
- Versionamento de dados;
- Log de atualizações;
- Persistência de dados em arquivos, com suporte a formatos XML e binários;
- Delayed updates para campos Auto incremento.

Arquitetura em camadas

A arquitetura do AnyDAC é composta de múltiplas camadas, fracamente acopladas. Cada camada provê serviços que podem ser solicitados por outras camadas, uma vez que a API de serviços é toda definida como uma interface interoperável. A **Figura 1**, extraída do site oficial do produto, apresenta a ilustração desta arquitetura, mostrando as camadas envolvidas.

Para fins de entendimento, a seguir é exposta uma breve explanação sobre cada uma destas camadas.

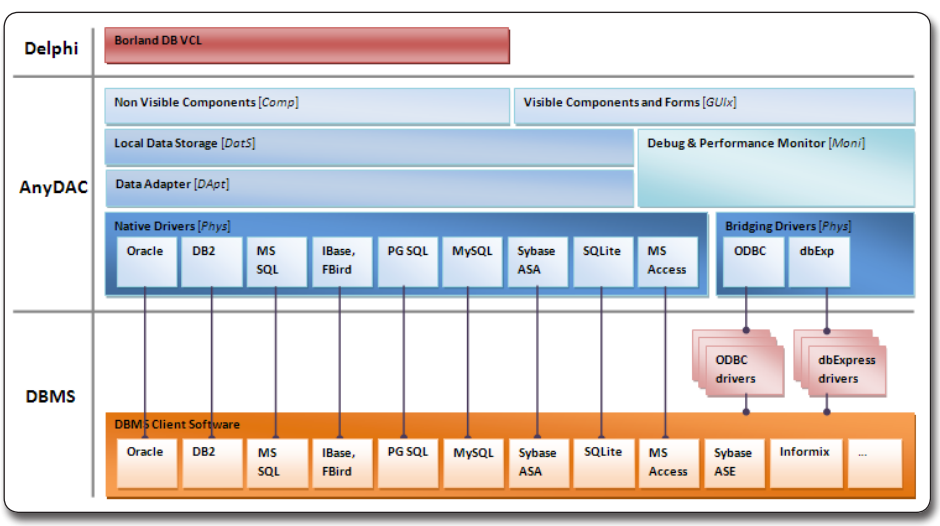

Figura 1. Visão geral da arquitetura do AnyDAC

Non Visible Components [Comp]

Conforme seu nome próprio sugere, a camada Non Visible Components engloba os componentes não visuais do AnyDAC, ou seja, aqueles não visíveis em Runtime para o usuário final. Isso inclui elementos como TADConnection, TADQuery e TADStoredProc.

Visible Components and Forms [GUIx]

Esta camada engloba os componentes utilizados para prover a interação com o usuário final em uma aplicação. Dessa forma, há implementações para todas as plataformas suportadas pelo AnyDAC – VCL (Visual Component Library), LCL (Lazarus Component Library) e FireMonkey. Como exemplo, é possível citar as caixas de diálogo e de espera para determinadas operações de banco de dados. Essa interatividade é provida por elementos como TADGUIxLoginDialog e TADGUIxWaitCursor.

Local Data Storage [DatS]

Local Data Storage é o nome da camada que remete ao mecanismo de dados em memória do AnyDAC. Sua implementação é semelhante à estrutura de DataSet do ADO.NET, que se estabelece a partir da combinação de vários objetos relacionados como DataTable e DataRow.

Data Adapter [DApt]

Esta camada se incumbe de automatizar as operações de leitura (read) de result sets oriundos de consultas de dados, bem como a aplicação das atualizações (update) destes dados de volta ao banco. Neste cenário, incluem-se operações complexas, tais como as associadas a relações aninhadas (Nested) e Mestre-Detalhe.

Nota do DevMan 3

ADO.NET remete a um conjunto de classes presentes no Microsoft .NET Framework, direcionadas fundamentalmente para a parte de acesso a dados de uma aplicação, utilizando para isso objetos especiais, tais como os do tipo DataSet. Dentre suas características se destaca seu modelo "desconectado", que otimiza o cenário das aplicação de banco de dados no lado cliente, bem como seu suporte a inúmeros SGBDs.

Debug & Performance Monitor [Moni]

Tal como seu nome sugere, Debug & Performance Monitor é a camada do AnyDAC voltada à depuração e monitoramento de performance. Sendo assim, abrange toda sua capacidade de monitoramento e rastreamento das interações estabelecidas entre a aplicação e o SGBD, refletida em componentes como TADMoniRemoteClientLink e TADMoniFlatFileClientLink.

Native Drivers [Phys]

Native Drivers abrange a coleção de drivers "nativos" de acesso a cada SGBD suportado pelo AnyDAC. As características específicas de cada SGBD são então "adaptadas" para a API do próprio AnyDAC a partir das recomendações de cada fornecedor.

Bridging Drivers [Phys]

Em complemento à Native Drivers, que provê toda a parte de drivers nativos para acesso direto aos diversos SGBDs suportados pelo AnyDAC, a camada Bridging Drivers engloba a parte de drivers "bridge". Estes drivers se utilizam então de APIs genéricas de acesso a dados, tais como ODBC e dbExpress, para se conectar a SGBDs.

Componentes

A biblioteca de componentes do AnyDAC é bastante extensa. Uma vez instalada, ela se anexa à Tool Palette do Delphi, de forma a compor novos grupos de elementos. Citando como exemplo a VCL, no total são cinco novos grupos – AnyDAC, AnyDAC UI, AnyDAC Links, AnyDAC Services e AnyDAC Devs – conforme mostra a **Figura 2**.

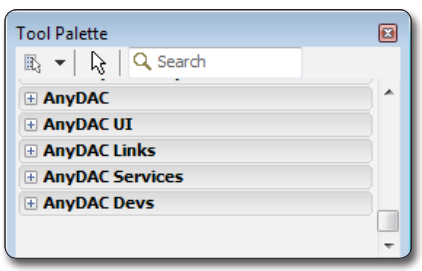

Figura 2. Grupos de componentes AnyDAC para VCL

Em razão da vasta quantidade de componentes, a apresentação de todos se tornaria muito maçante e pouco usual, sendo assim, a seguir serão expostos alguns dos mais usuais. Neste momento, poderá ser observada a proximidade destes com componentes nativos conhecidos, tais como os das guias InterBase (IBX), dbExpress, dbGo e BDE da Tool Palette do Delphi. Os componentes a seguir se referem ao grupo AnyDAC, e são tratados como AnyDAC Core Components.

TADConnection

Componente designado para estabelecer a conexão com um SGBD. Componentes nativos similares: TSQLConnection, TADOConnection, TIBDatabase e TDatabase.

TADTransaction

TADTransaction é a classe do componente responsável pelo gerenciamento de transação de uma conexão. Sua intenção é similar ao componente TIBTransaction, da guia InterBase.

TADQuery

Componente capaz de executar consultas SQL no servidor de banco de dados. Componentes nativos similares: TSQLQuery, TSQLDataSet, TADOQuery, TIBQuery e TQuery.

TADStoredProc

Componente a ser utilizado para se executar procedimentos armazenados (Stored Procedures) no servidor de banco de dados. Componentes nativos similares: TSQLStoredProc, TADOStoredProc, TIBStoredProc e TStoredProc.

TADScript

Componente que implementa um mecanismo de execução de scripts SQL, tal como o TIBScript da guia InterBase.

TADUpdateSQL

Componente responsável por aplicar atualizações, associadas a consultas (Query) e Stored Procedures, no banco de dados. Componentes nativos similares: TUpdateSQL e TIBUpdateSQL.

Além destes, a maioria das aplicações que utilizam AnyDAC irá necessitar de um componente "Driver Link", referente ao SGBD utilizado, para que o driver adequado possa ser vinculado e configurado na aplicação. Tais componentes ficam localizados no grupo AnyDAC Links na Tool Palette (**Figura 3**). Pela própria nomenclatura de cada um, é possível apontar sua especificação. Como exemplo, TADPhysIBDriverLink está relacionado aos SGBDs InterBase e Firebird, assim como TADPhysOracleDriverLink é direcionado a bancos Oracle. De forma complementar, neste mesmo grupo há ainda componentes mais "genéricos", tal como TADPhysODBCDriverLink, para vinculação de drivers ODBC à aplicação.

Já o grupo AnyDAC Services contempla os componentes relacionados a serviços de banco de dados. Nesta linha, serviços de Backup e Restore são os mais conhecidos e comumente utilizados. Para estes casos, o AnyDAC novamente segmenta os componentes de acordo com o SGBD, tais como TADIBBackup e TADIBRestore, que são específicos a bancos InterBase/Firebird.

Por fim, AnyDAC UI (**Figura 4**) é o grupo que abrange os componentes da biblioteca, voltados à interação com o usuário:

- TADGUIxLoginDialog: exibe uma caixa de diálogo ao usuário, contendo campos que deverão ser preenchidos com as informações das credenciais (usuário e senha) do banco de dados;
- TADGUIxErrorDialog: exibe uma janela de diálogo onde o usuário final é notificado sobre exceções geradas pelo AnyDAC;
- TADGUIxScriptDialog: exibe uma caixa de diálogo que mostra o progresso de execução de um SQL. Este recurso proporciona ao usuário final uma melhor visualização da execução do script, possibilitando a identificação do trecho que está sendo executado naquele momento, bem como o cancelamento do processo;
- TADGUIxWaitCursor: controla o cursor de espera (wait cursor) sobre as ações efetuadas numa aplicação AnyDAC;
- TADGUIxAsyncExecuteDialog: exibe uma caixa de diálogo que mostra o progresso de execução de uma consulta SQL. Através dela, o usuário final poderá tanto visualizar o processo quanto cancelá-lo;

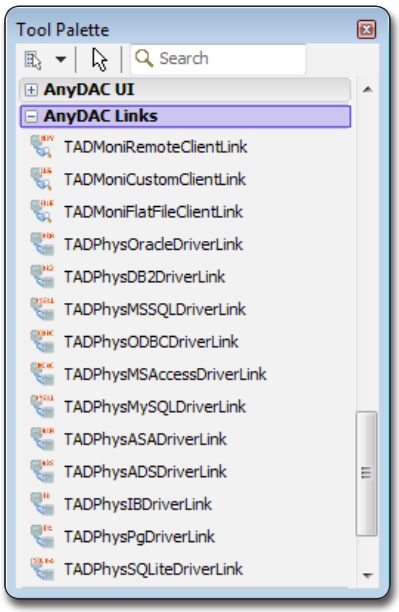

Figura 3. Componentes AnyDAC Links

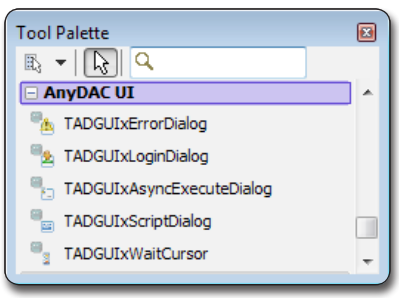

Figura 4. Componentes AnyDAC UI

Criando a primeira aplicação com AnyDAC

De forma a contribuir com o aprendizado do assunto, iremos agora ir além da teoria, utilizando a prática para reforçar os conceitos apresentados até agora. Como todo assunto novo abordado em desenvolvimento de software, a primeira medida será demonstrar a utilização mais básica dos elementos envolvidos, que neste caso, são os próprios componentes da biblioteca. Dito isto, será construída então uma aplicação simples, do tipo VCL Forms, que fará uso dos componentes AnyDAC para se conectar a um banco de dados InterBase (InterBase XE3) e obter os dados de uma de suas tabelas. A seguir, tais dados serão exibidos em tela, por meio de um controle DBGrid. A escolha pelo InterBase se justifica pelo fato do mesmo fazer parte do nicho de produtos da Embarcadero, além de atualmente acompanhar a instalação do próprio Delphi XE3. Para agilizar ainda mais este desenvolvimento, será utilizado também o banco de exemplo que acompanha o próprio SGBD, denominado Employee.

Partindo de um novo projeto VCL Forms Application, a primeira ação a ser tomada é a adição de um container (Data Module) para centralizar todos os eventuais componentes de acesso a dados da aplicação. Isso garante uma melhor organização do projeto, além de ser uma boa e conhecida prática no cenário Delphi. Nesta estrutura, será adicionado então o primeiro componente AnyDAC do projeto, TADConnection, que ficará responsável por estabelecer a conexão com o referido banco de dados.

Conectando ao banco de dados

Traçando um paralelo com o dbExpress, a forma de se estabelecer uma nova conexão com um banco de dados no AnyDAC torna-se muito semelhante. Toda a definição dos parâmetros desta é realizada na propriedade Params do componente. Dito isto, a **Listagem 1** mostra o código do arquivo .dfm relacionado ao componente, já com as configurações direcionadas ao exemplo.

Pelo código apresentado, podemos ver inicialmente a definição da propriedade ConnectionName que assim como no dbExpress, nada mais é do que uma nomenclatura dada à conexão. A seguir, há a lista de parâmetros da conexão, dentre os quais se destacam o caminho do arquivo do banco (Database)

e a indicação do Driver AnyDAC (DriverID) para InterBase/Firebird. No trecho, o caminho mostrado se refere ao caminho padrão do banco de exemplo Employee, numa instalação em um ambiente em inglês.

Buscando dados de uma tabela

Com o AnyDAC, para se realizar uma consulta de dados sobre uma determinada tabela do banco e recuperar essas informações na aplicação, utiliza-se um componente TADQuery. Uma vez adicionado ao Data Module do exemplo, ele se liga ao componente TADConnection, tal como a relação TSQLConnection e TSQLQuery no dbExpress. Ainda seguindo esta comparação inicial entre frameworks, o local onde será determinado o comando SQL no componente do AnyDAC também se dá numa propriedade denominada SQL. A seguir na **Listagem 2** é exposto o comando SQL a ser utilizado.

Listagem 1. Configurações do componente TADConnection

```
01 object ADConnection1: TADConnection
02   ConnectionName = 'Employee'
03   Params.Strings = (
04    'DriverID=IB'
05    'Database=C:\Program Files\Common Files\CodeGear Shared\Data\
06      EMPLOYEE.GDB'
07    'Protocol=TCPIP'
08    'User_Name=sysdba'
09    'Password=masterkey')
10   Connected = True
11   LoginPrompt = False
12   ...
13 end
```

Listagem 2. Instrução SQL utilizada

```
select
  CUST_NO,
  CUSTOMER,
  PHONE_NO,
  ADDRESS_LINE1,
  ADDRESS_LINE2,
  CITY,
  COUNTRY
from
  CUSTOMER
```

Seguindo a proposta inicial, o próximo passo é expor ao usuário final os dados buscados do banco. Dessa forma, voltando ao fluxo tradicional de desenvolvimento de aplicações de banco de dados no Delphi, resta adicionar ao Data Module um componente TDataSource, que fará o papel de intermediador entre o DataSet (TADQuery) e o Grid da UI a ser construída. Para a elaboração desta última, uma sugestão é mostrada na **Figura 5**.

Neste momento, ao efetuar um teste ainda em tempo de Design, através da ativação (Connected = True) do componente TADQuery, temos os dados sendo devidamente mostrados no Grid (**Figura 6**). No entanto, ao tentar executar a aplicação, uma nova particularidade do AnyDAC é apresentada, conforme mostra a **Figura 7**. Uma exceção é levantada informando, essencialmente, que o driver utilizado em questão (IB) não está registrado. Todavia, a própria mensagem nos provê uma solução, indicando que para efetuar este registro, basta adicionar ao projeto um componente do grupo AnyDAC Links, relacionado ao SGBD ao qual se está trabalhando, neste caso, TADPhysIBDriverLink para InterBase.

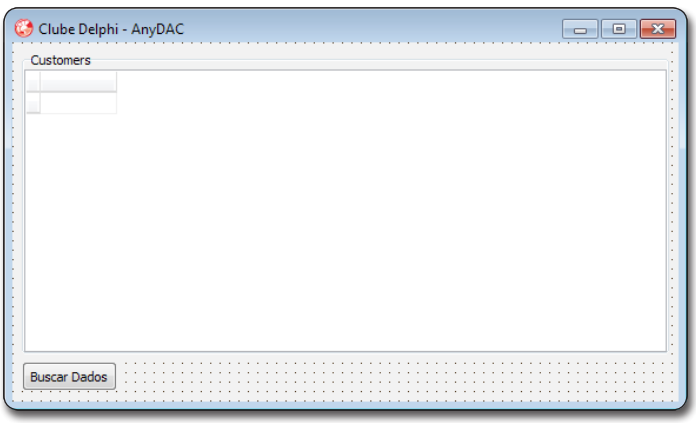

Figura 5. Formulário da aplicação de exemplo

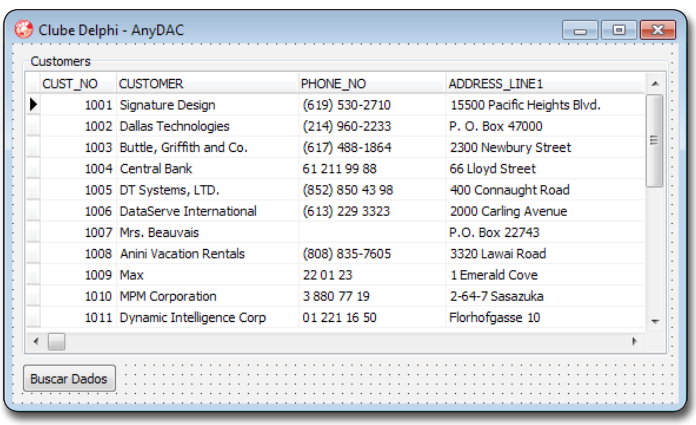

Figura 6. Dados sendo mostrados em tempo de Design

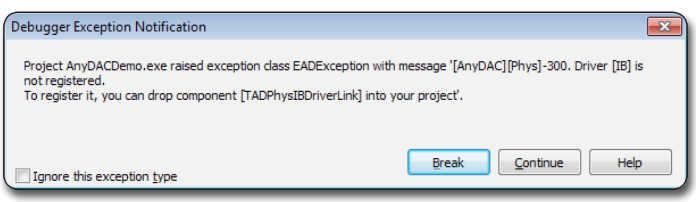

Figura 7. Exceção levantada pelo AnyDAC, relacionada a driver

Ao se tentar executar a aplicação novamente, mais uma particularidade do AnyDAC é apresentada (**Figura 8**). Da mesma forma que antes, a mensagem apresentada é autoexplicativa, e indica a inclusão de um componente TADGUIxWaitCursor ao projeto. Por conseguinte, ao realizar este procedimento, enfim a aplicação está apta a operar em Runtime. Somente para registro, o código para o evento OnClick do botão é apresentado a seguir na **Listagem 3**.

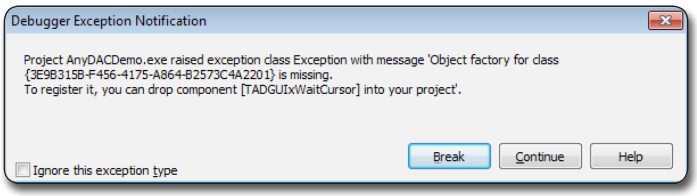

Figura 8. Exceção levantada pelo AnyDAC, relacionada a WaitCursor

Listagem 3. Evento OnClick do botão

```
procedure TForm1.btnBuscarDadosClick(Sender: TObject);
begin
  DBGrid1.DataSource.DataSet.Open;
end;
```

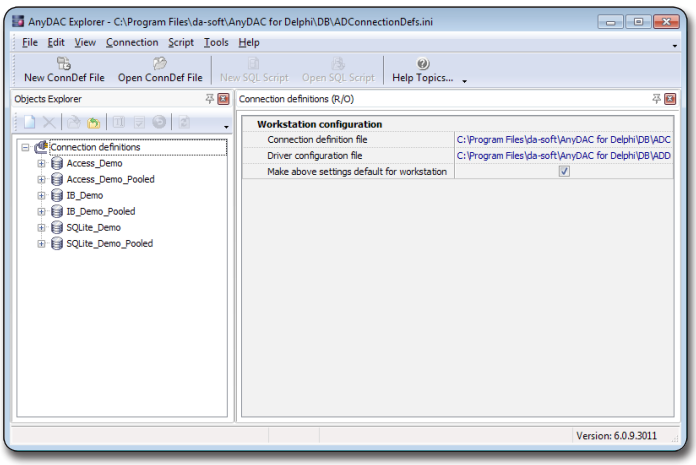

Figura 9. AnyDAC Explorer

Manipulando dados

Em vista da comparação realizada com o dbExpress, é essencial ressaltar uma característica importante do componente DataSet (TADQuery) do AnyDAC. Isto porque, conforme visto, foi possível realizar o seu vínculo com o controle em tela (DBGrid) sem a necessidade de utilização de componentes intermediários, tais como os usuais TDataSetProvider e TClientDataSet. Isso se deve ao fato de sua característica bidirecional, oposta aos DataSets do dbExpress, que são unidirecionais e necessitam de elementos intermediários como complemento, no que diz respeito, por exemplo, à navegação de dados. Ainda nesta mesma linha de raciocínio, os componentes DataSets (TADQuery, TADTable, TADStoredProc e TADMemTable) do AnyDAC ainda apresentam a capacidade de manipulação efetiva das informações de banco, refletida nas operações de inserção, edição e exclusão de dados.

Com a finalidade de demonstrar tais características, iremos evoluir o exemplo apresentado, para que a aplicação contemple os recursos de inserção, edição e exclusão de dados da tabela Customer. Seguindo por uma linha mais didática, a adição de um controle TDBNavigator ao formulário, ligado ao TDataSource relacionado ao TADQuery já seria o suficiente. Feito isso, ao se efetuar as operações de inclusão, edição e exclusão sobre os dados mostrados no Grid, as ações são prontamente refletidas na tabela do banco de dados. De forma programática, isso também poderia ser feito utilizando-se os métodos padrões de um TDataSet tradicional, tais como Append, Insert, Edit, Delete e Post, uma vez que o AnyDAC compatibiliza todos os seus DataSets.

Utilitário ADExplorer

A instalação do AnyDAC provê ainda ao ambiente uma série de utilitários para os mais variados fins. Dentre estes, o AnyDAC Explorer (ou simplesmente ADExplorer) se destaca, sendo um utilitário similar aos já habituais do Delphi, como BDE Database Explorer e Data Explorer. Sendo assim, é o local onde se centraliza toda a parte de definição de conexões aos bancos de SGBDs suportados, possibilitando inclusive a execução de scripts SQL sobre estes. No IDE, o ADExplorer é disponibilizado no menu específico criado pela biblioteca (AnyDAC) e seu layout pode ser visto na **Figura 9**.

Dentre suas possibilidades, a seguir são listadas algumas das operações a serem realizadas no ADExplorer:

- Criação e manipulação de arquivos e definições de conexão;
- Criação e manipulação de scripts SQL relacionado a um banco de dados;
- Visualização e navegação sobre os objetos do banco de dados, tais como tabelas, campos, Strored Procedures e Triggers;
- Visualização e manipulação dos dados de uma tabela;

Aqui surge então mais uma opção para se realizar a conexão com um banco de dados pelo AnyDAC, agora através do uso de um arquivo de conexão. Numa situação semelhante ao uso do dbxconnecions.ini do dbExpress, aqui o desenvolvedor faz uso do ADExplorer para a definição de um arquivo semelhante, tratado como Connection Definition File (arquivo de definição de conexão). Por padrão, o AnyDAC já traz consigo um modelo para este fim, nomeado como ADConnectionDefs.ini, localizado na pasta DB de seu diretório de instalação. Uma vez que é definida uma conexão neste arquivo, esta pode ser referenciada no componente TADConnection, substituindo a abordagem utilizada anteriormente no exemplo.

Conectando a um banco de dados utilizando um Connection Definition File

De forma prática, para a efetiva utilização de uma conexão definida em arquivo, o passo inicial é a configuração da mesma no arquivo ADConnectionDefs.ini, seja de forma manual, ou através do ADExplorer. A **Listagem 4** mostra um exemplo da definição dos parâmetros de conexão para o banco Employee utilizado anteriormente.

O AnyDAC, além do arquivo de definição de conexão, traz ainda outro arquivo externo, denominado ADDrivers.ini, similar ao arquivo dbxdrivers.ini do dbExpress e que se refere à configuração dos drivers das conexões utilizadas. Numa exemplificação simples, para esta situação, uma configuração válida seria a indicação explícita do caminho da DLL cliente do SGBD utilizado (InterBase). Dito isto, a seguir na **Listagem 5** é mostrada a referida configuração.

Listagem 4. Parâmetros de conexão para o banco Employee

```
01 [Employee-IB]
02 DriverID=IB
03 DataBase=C:\Program Files\Common Files\CodeGear Shared\Data
     \EMPLOYEE.GDB
04 User_Name=SYSDBA
05 Password=masterkey
```

Listagem 5. Definição do arquivo de Drivers

```
01 [IB]
02 VendorLib=C:\Program Files\Embarcadero\InterBase\bin\gds32.dll
```

Neste tipo de abordagem, para que o componente TADConnection passe a utilizar a conexão previamente definida no arquivo externo padrão, descartando assim a necessidade de se editar manualmente sua propriedade Params, basta atribuir o nome da conexão (Employee-IB) à sua propriedade ConnectionDefName, tal como mostra a **Figura 10**.

Agora, sua propriedade Params irá apenas apresentar a indicação da **Listagem 6**.

Visualizando objetos de banco de dados

Conforme mencionado, através do ADExplorer é possível se visualizar os objetos do banco de dados relacionado a uma conexão. Dessa forma, ao se abrir o arquivo ADConnectionDefs.ini em sua

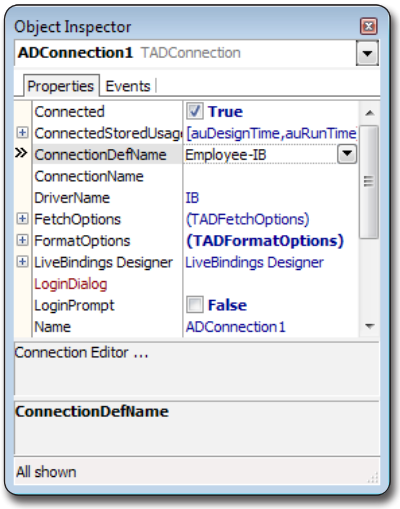

Figura 10. ADConnection - ConnectionDefName

Listagem 6. Propriedade Params da conexão

```
01 object ADConnection1: TADConnection
02    Params.Strings = (
03     'ConnectionDef=Employee-IB')
04    LoginPrompt = False
05 ...
06 end
```


FÓRUM
DEVMEDIA
O lugar perfeito para você ficar por dentro de tudo o que acontece nas tecnologias do mercado atual

No novo Fórum da DevMedia você vai encontrar canais específicos de Delphi, ASP.NET, Java, Banco de Dados e Engenharia de Software; além de ter contato com profissionais qualificados da área para troca de informações, sugestões e muito mais.

ACESSE AGORA
www.devmedia.com.br/forum

DevMedia

estrutura, é habilitada a navegação de todas as conexões definidas no arquivo. A expansão do nó de uma conexão faz com que toda a estrutura do banco de dados seja carregada para a estrutura TreeView, tal como mostra a **Figura 11**. Este recurso já é bem familiar para a comunidade Delphi, devido ao uso do Data Explorer.

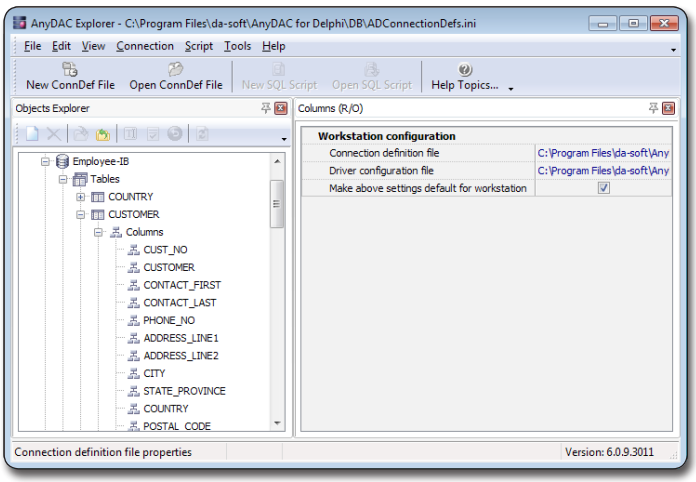

Figura 11. Visualizando objetos de banco de dados no ADExplorer

Utilitário ADMonitor

O AnyDAC Monitor (ADMonitor) é outro utilitário provido pela instalação do AnyDAC. Conceitualmente falando, ele pode ser definido como sendo uma ferramenta voltada para a depuração da aplicação, isto porque, dentre seus principais recursos, está o de rastreamento (Tracing) da comunicação entre a aplicação e o SGBD. Sendo assim, através dele o desenvolvedor é capaz de monitorar e obter um registro detalhado desta comunicação.

No IDE, o ADMonitor apresenta-se sob o nome de Monitor no menu AnyDAC. Quando acionado, sua estrutura é apresentada em tela conforme mostra a **Figura 12**. Ainda relacionado ao Trace, a saída produzida pelo processo pode ser controlada por diferentes componentes do AnyDAC, dando ao desenvolvedor diferentes formas de visualização dos resultados obtidos. Para que estes sejam direcionados à estrutura do ADMonitor, o projeto da aplicação deverá conter um componente TADMonitorRemoteClientLink. Além desta, há ainda outras duas opções. A primeira se faz pela utilização de um componente TADMoniFlatFileClientLink, que direciona a saída do processo para um arquivo de texto. Por fim, outra alternativa inclui o componente TADMoniCustomClient-Link que direciona a saída para um manipulador de eventos personalizado.

Conclusão

O anúncio feito pela Embarcadero sobre a aquisição do AnyDAC gerou certa polêmica em meio à comunidade Delphi. Isto porque, muitos ficaram temerosos sobre o real futuro do dbExpress e até que ponto esta nova biblioteca de componentes poderia "substituí-lo", afetando diretamente projetos legados e correntes que se utilizam da solução nativa. Polêmicas à parte, a situação atual é que o desenvolvedor Delphi passa a contar com mais uma opção no que diz respeito à parte de acesso a dados em seus projetos. Indo além, é possível ainda fazer a mescla de tecnologias, uma vez que o AnyDAC provê suporte ao próprio dbExpress, bem como a servidores DataSnap (cuja arquitetura é baseada no dbExpress).

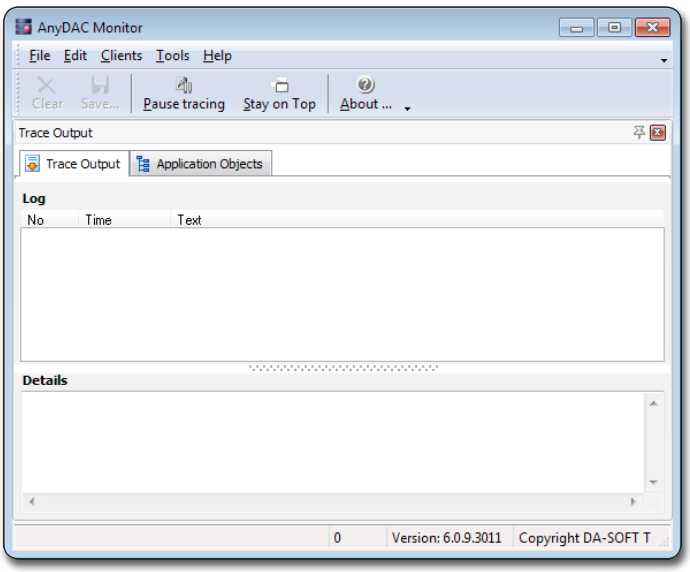

Figura 12. AnyDac Monitor (ADMonitor)

Em seu histórico recente, o Delphi vem proporcionando novos desafios à comunidade, fruto de sua constante evolução. Só para citar, nos tempos atuais pudemos acompanhar a inclusão de novas tecnologias a seu contexto, tais como AQTime, Final Builder, CodeSite, Subversion, FireMonkey, FastReport e agora AnyDAC.

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador Delphi há 8 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor.

Delphi XE3 – Download Trial
https://downloads.embarcadero.com/free/delphi

Delphi XE3 – Página oficial do produto
http://www.embarcadero.com/products/delphi

AnyDAC - Página oficial do produto
http://www.da-soft.com/anydac

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Melhore seu Código com Domain Driven Design

Codifique pensando nas regras de negócio

O Domain-Driven Design foi criado por Eric Evans e retratado em seu livro "Domain-Driven Design: Tackling Complexity in the Heart of Software", onde ele fala extensivamente sobre os princípios que envolvem a ideia do DDD e como eles podem ser aplicados e suportados em código, mostrando de forma detalhada a solução para problemas que ocorrem a mais de vinte anos.

O que é importante?

O DDD diz que o mais importante em um software não é o seu código, nem sua arquitetura, nem a tecnologia sobre a qual foi desenvolvido, mas sim o problema a ser resolvido, ou seja, a regra de negócio. Se pararmos para pensar isso realmente faz sentido, pois sem ela não existe razão do software existir. É por isso que devemos focar nosso tempo e atenção nessa regra, mais do que em qualquer outra coisa. A complexidade de um sistema, em quase 100% dos projetos não está na tecnologia utilizada, mas no negócio envolvido. Esse negócio é chamado de "coração" pelo livro do Eric Evans, é o centro do aplicativo e todo o restante deve ser construído para atender esse centro da melhor forma possível.

Para deixar esse ideia bem clara, vamos imaginar um sistema de caixa de supermercado que você foi incumbido de desenvolver. Com quem você irá tirar suas dúvidas sobre o processo de funcionamento? O dono da rede? Os clientes que compram no supermercado? Eles podem ter uma noção de funcionamento, mas, quem conhece a fundo, é quem está envolvido diretamente com as rotinas de caixa, ou seja, os caixas desse supermercado. Agora vamos fazer uma análise do negócio à luz do DDD.

A rotina e processos envolvidos no caixa são chamados de domínio da aplicação. Este é o cerne da aplicação, o problema a ser solucionado. Durante o desenvolvimento temos a tendência a tirar nossa atenção disso e a colocar em outro lugar, por exemplo, em aspectos técnicos. Assim, perdemos o conhecimento mais aprofundado das regras de negócio, consequentemente, perdemos qualidade no produto final pois como não conhecemos adequadamente o negócio, como nosso sistema irá atender também de forma adequada as necessidades do cliente? O que precisa ficar claro para nós é que, não basta ter só conhecimento técnico, utilizar as mais novas ferramentas e frameworks, mas

Resumo DevMan

Melhore seu Código com Domain Driven Design:

Durante o ciclo de vida de um projeto, várias alterações de regras de negócio acontecem. Muitas vezes é preciso responder a essas mudanças de forma muito rápida e, sem pensar, acabamos codificando algo que resolve o problema na hora, mas que na verdade, não foi o melhor código. Com a correria do dia não voltamos a rever esse trecho do sistema e o deixamos lá. Esse é um cenário comum para pesquisas SQL. Para resolver um problema rapidamente, escrevemos uma sentença que já existe em algum lugar do sistema, ou seja, replicamos essas instruções e esquecemos de corrigir isso. Com o tempo só vamos lembrar dessa sentença quando a mesma precisar ser alterada, então teremos que procurar por todo o sistema. O pior é que isso irá aumentar sempre, até que uma simples correção de alguma pesquisa SQL se torne um desafio a ser corrigido em vários locais. Com regra de negócio espalhada por vários locais, fica difícil o entendimento do sistema por outro desenvolvedor e até mesmo pelo próprio criador do projeto. Com DDD podemos organizar nosso código, mesmo procedural, de forma elegante, pensando nas regras de negócio e não em tecnologia.

Em que situação o tema é útil:

Existem alguns termos que passam despercebidos pela maioria da comunidade Delphi, uma vez que a ferramenta favorece um estilo de programação RAD, onde se desenvolve muito rápido e quase sempre, a arquitetura não é abordada como deveria. Por isso, muitos nunca ouviram falar sobre Domain-Driven Design, é importante ressaltar que não se trata de um novo framework nem uma nova linguagem de programação, mas sim de uma nova abordagem (alguns diriam até um novo paradigma) para o desenvolvimento de software, ou seja, uma série de ideias e princípios que ao serem aplicados no processo de desenvolvimento de software podem simplificar o mesmo e agregar mais valor ao produto final.

é preciso que todos da equipe de desenvolvimento tenham o entendimento sobre o problema a ser resolvido, pois é isso que o cliente espera.

Todos falando a mesma língua

Quando uma equipe de desenvolvimento é escalada para implementar um sistema, é quase certeza que inicialmente, nenhum deles conheça o domínio do problema. Contudo, com o passar do tempo, troca de informações com especialistas, esse conhecimento é adquirido. Isso vem através da boa comunicação, mas para isso acontecer, todos devem falar a mesma língua. Mas, o que isso significa?

Para entender, voltemos ao sistema de caixa citado anteriormente. Digamos que estamos coletando requisitos junto a um operador de caixa e o mesmo diz que a cada duas horas é realizada uma sangria. Se nunca tivéssemos conhecido nada sobre operações de caixa, por acaso saberíamos o que é uma sangria? Dificilmente não. Então seria necessário pedir mais explicações sobre o que seria isso. Esse é o ponto. Cada negócio possui seus próprios jargões, peculiaridades que seus especialistas conhecem, mas os desenvolvedores não. O mesmo pode acontecer de forma oposta, o usuário vem lhe perguntar como ele entrará no sistema e você começa a explicar que tudo será feito por Active Directory, quando o que na verdade bastaria informar: "por um usuário e senha". Acredito que um usuário comum sairia com essa resposta com mais dúvidas do que quando chegou.

Para minimizar essa discrepância, o DDD possui o conceito de linguagem ubíqua. A ideia é construir uma espécie de vocabulário que torne a comunicação entre desenvolvedores e especialistas, a mais natural possível. Para isso, pode ser necessário abstrair alguns conceitos, sejam de negócio ou técnicos, e compartilhar com os envolvidos, em outra situação pode ser necessário dar mais detalhes sobre um determinado assunto. Tudo isso fica documentado para conhecimento de todos e consequentemente, todos entenderão o que é dito.

Cada coisa em seu lugar

Para começar a aplicar o DDD é preciso saber que o mesmo divide um software em camadas lógicas. Aos realizar essa divisão é possível separar responsabilidades e focar nas regras de negócio. A **Figura 1** mostra a divisão proposta pelo DDD.

Se olharmos novamente no cenário apresentado, vemos que podemos ter lógica de negócio espalhada pelas diversas camadas do sistema, isso deve ser evitado ao máximo, para evitar complexidade de alteração. O grande problema é que alterar alguma dessas regras se torna um verdadeiro pesadelo, uma vez que o desenvolvedor precisará literalmente rastrear todos os locais onde ela deve ser modificada. Para evitar esse tipo de problema, devemos isolar todo o código responsável por controlar o domínio de maneira que ele tenha uma única responsabilidade: implementar regras de negócio. Essa camada deve ser o principal local modificado quando alguma regra da aplicação tiver que ser alterada. Vejamos cada camada:

- **Domain Layer:** Como já comentado, essa camada concentra toda a regra de negócio da aplicação. Essa deve ser a sua única preocupação, delegando qualquer outro tipo de atividade para as demais camadas;
- **Infrastructure Layer:** É a camada de mais baixo nível, responsável por prover serviços como persistência e envio de e-mail, ou seja, dar o suporte tecnológico para as demais camadas;
- **Application Layer:** Coordena as atividades da aplicação, porém, não contém regras de negócio. Pode, por exemplo, manter um estado de determinada transação. Essa é a camada mais discutida, pois dependendo do domínio da aplicação, ela pode não ser necessária.
- **User Interface Layer:** Responsável pela interação com o usuário, seja interpretando comandos ou exibindo informação para o mesmo.

Veremos agora os principais padrões.

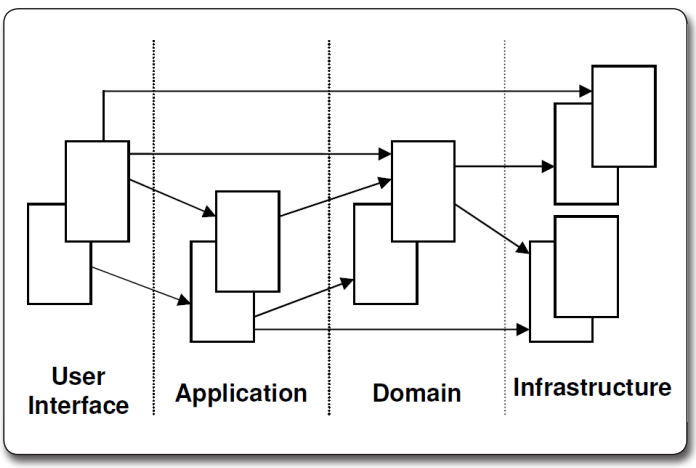

Figura 1. Arquitetura padrão sugerida pelo DDD

Organizando o domínio

O domínio é uma coleção de ideias, conceitos, interpretações de situações do mundo real. Tudo isso deve ser transformado em código que atenda às necessidades dos usuários. Esse código é a camada chamada de modelo. Ela abstrai as regras envolvidas no negócio, formando um guia para o desenvolvimento. E é esse o grande desafio, criar um modelo que represente de forma clara, objetiva e útil ao desenvolvimento os conceitos do domínio. Ele deve expressar a realidade e ser de fácil implementação. Para auxiliar nesse processo o DDD sugere algumas práticas, que são a construção de alguns conceitos específicos, como mostra a **Figura 2**.

Classificando objetos

Por mais que em um desenvolvimento tradicional, procedural, não temos o hábito de criar classes, é possível ainda realizar um comparativo entre alguns tipos de objetos que o DDD define. São eles: entidades, objetos de valor, serviços e agregações.

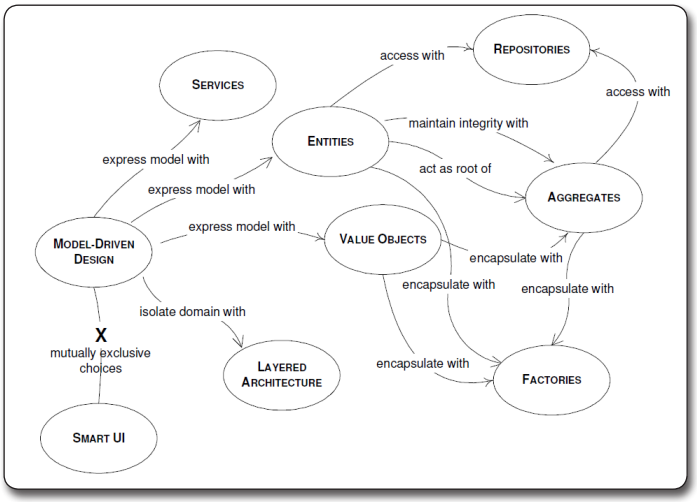

Figura 2. Diagrama que mostra o relacionamento entre os principais padrões do DDD

• **Entidades**: são objetos identificados por sua identidade e não por seus atributos. O exemplo mais simples é uma pessoa. A representação de uma pessoa em software geralmente conta com atributos como nome, data de nascimento, CPF, RG etc. Porém, como podemos distinguir cada pessoa unicamente? Pelo nome e data de nascimento com certeza não, pois muitas pessoas nascem em um mesmo dia e podem existir pessoas com o mesmo nome também. CPF e RG podem ser usados, se levarmos em conta apenas cidadãos brasileiros. A principal ideia é que cada pessoa deve ser única, não sendo possível criar uma pessoa igual apenas "copiando" os atributos de outra. Esse tipo de objeto costuma ser persistente, ou seja, precisa ser salvo para ser recuperado posteriormente. Em um sistema centrado a dados, podemos dizer que uma entidade é a representação de um tabela do banco de dados.

• **Objetos de Valor**: são objetos que não possuem uma identidade definida, eles são definidos apenas por seus atributos. Para o domínio, não é importante que sejam identificados. O que importa é somente o valor que carregam. Imagine aqui aquela lista de valores fixa que você carrega em um ClientDataSet temporário, apenas para exibição de alguma informação. Por exemplo, tipo de pessoa: Física/Jurídica. Esse é um exemplo bem simples mas real.

• **Serviços**: são objetos que fornecem um comportamento que não se adapta a nenhuma entidade ou objeto de valor, mas é um conceito importante no contexto do domínio. Um exemplo é o cálculo de despesa de envio de produto em um site de compras. O CEP é utilizado para efetuar esse cálculo, mas essa lógica está além do objetivo do objeto CEP, portanto não é correto que fique no mesmo um método "CalcularFrete" por exemplo, ou seja, não é indicado. Entretanto, esse conceito também não está diretamente relacionado a nenhum outro objeto. O melhor nesse caso é optar por um Serviço cujo único objetivo é efetuar o cálculo. Serviços geralmente terão como nome um verbo ("CalcularFrete", por exemplo), diferente de entidade e objetos de valor, cuja descrição é geralmente um nome.

• **Agregação**: Este está relacionado ao conceito de mestre-detalhe, pai-filho. Objetos que fazem parte de uma agregação representam um agrupamento, um conceito só. Assim, um objeto filho não tem sentido de existir sem seu objeto pai. O exemplo clássico dessa situação é nota fiscal e seus itens. Os itens só existem enquanto existe a nota, se a mesma deixar de existir, seus itens não precisam permanecer.

Temos vários conceitos, mas nesse artigo vamos tratar do problema de espalhar sentenças SQL em todo lugar. Isso pode ser resolvido pelo conceito de Repositórios.

Centralizando

Um repositório abstrai o acesso à fonte de dados, provendo uma interface similar a uma coleção que pode ser facilmente manipulada pelo sistema. Esse padrão é um dos principais fundamentos do DDD, pois é através dele que a camada de domínio fica livre de qualquer código necessário para o acesso à fonte de dados. Podemos utilizá-lo para criação de sentenças SQL, de forma centralizada. Dessa forma quando for necessário alterar algo nessas sentenças, elas poderão ser facilmente localizadas.

Um exemplo

Para demonstrar o problema e enxergarmos a solução vamos utilizar um domínio levemente complexo. O controle de intimações que uma empresa de cobrança deve possuir. Esse controle envolve uma dívida, que possui um ou mais devedores e esses, possuem seus endereços. Para cada devedor deve ser realizada uma intimação, que deve ter sua entrega registrada. Cada intimação deve possuir três tentativas de entrega. Quem entrega uma intimação é o entregador, que recebe valores diferenciados por cada intimação, baseados no recebimento ou não da intimação. Dessa forma, podemos modelar o seguinte banco de dados exibido na **Figura 3**.

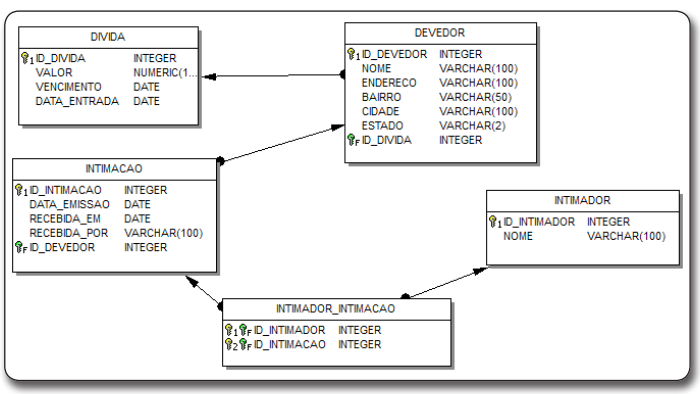

Figura 3. Banco de dados de controle de intimações

Criamos um novo projeto como de costume, acessamos o banco de dados através do dbExpress, criamos então os objetos TSQLQuery e TClientDataSet que irão armazenar e expor nossas entidades. Nesse exemplo elas são Dívida, Devedor e Intimação.

Realizamos seus cadastros básicos. Isso é o básico até o momento. Mas, digamos que uma intimação não pode ser emitida duas vezes, onde essa verificação é colocada?

Nota

A criação do projeto, bem como desenho das telas, controles de acesso a banco de dados, criação de units, não são mostrados aqui pois, o foco do artigo é a solução proposta. Contudo, o projeto encontra-se disponível para download.

É muito comum colocar essa regra de negócio, que é implementada através de uma consulta SQL, no próprio formulário onde ela é necessária. Seja porque foi mais rápido fazer ali, ou porque não se pensou que essa regra poderia ser utilizada em outro local. Temos então a primeira sentença "espalhada".

Outra regra poderia ser a de pagar um valor diferenciado por intimação recebida e não recebida. Teríamos uma tela onde se escolhe o intimador e se informa uma data, então seriam retornadas suas intimações e seu devido status, recebida ou não. Mais uma vez isso foi inserido nessa tela, prejudicando a reutilização.

Precisamos de um serviço?

As exigências da regra de negócio são a geração da intimação e a obtenção dos valores a pagar ao intimador. Isso pode ser centralizado em dois serviços, como mostra a **Listagem 1**.

Listagem 1. Serviços necessários

```
TImpressaoIntimacao = class
public
  procedure GerarIntimacoesPorData(Data: TDateTime);
end;

TGeracaoRepasseIntimador = class
public
  procedure GerarRepasse(IdIntimador: integer; InicioIntimacao: TDateTime;
  FimIntimacao: TDateTime);
end;
```

Os serviços estão isolando as regras de negócio e expondo funcionalidades através de métodos que possuem nomes relacionados à sua função. Isso facilita o entendimento de qualquer desenvolvedor. O interessante dos serviços é que eles fazem uso da infraestrutura existente, e não irão duplicar código. No caso de GerarIntimacoesPorData (**Listagem 2**), todo o acesso a dados será feito pelos seus respectivos repositórios.

Assim é possível ver que um serviço orquestra as funcionalidades existentes para atender um requisito e além disso, ele é isolado das outras camadas, para permitir sua reutilização. Na linha 5 o repositório de dívidas foi instanciado. Com isso temos acesso aos seus métodos de negócio, por exemplo, LocalizarDividaPorDataEntrada (linha 6). Não foi preciso aqui nesse momento escrever novamente o SQL de consulta, ele está encapsulado no repositório. Depois da linha 7 à linha 18, é feito uso da infraestrutura existente de acesso a dados. Como já foi mencionado, temos um DataModule que possui os componentes comuns (dbExpress + ClientDataSet) para acesso ao banco de dados, os repositórios os utilizam e os serviços também.

Fornecendo dados

É agora que o padrão Repository mostra sua utilidade. Ele diz que cada entidade possui seu repositório, com métodos de negócio que centralizam as operações. A **Listagem 3** mostra o repositório de dívidas, utilizado pelo serviço TImpressaoIntimacao.

Listagem 2. Método GerarIntimacaoesPorData

```
01 procedure TImpressaoIntimacao.GerarIntimacoesPorData(Data: TDateTime);
02 var
03 repoDivida: TDividaRepositorio;
04 begin
05  repoDivida := TDividaRepositorio.Create;
06  repoDivida.LocalizarDividaPorDataEntrada(Data);
07  while not dtm.cdsDivida.Eof do
08  begin
09   while not dtm.cdsDevedor.Eof do
10   begin
11    dtm.cdsIntimacao.Insert;
12    dtm.cdsIntimacaoDATA_EMISSAO.AsDateTime := now;
13    dtm.cdsIntimacaoID_DEVEDOR.AsInteger := dtm.cdsDevedorID_DEVEDOR.
      AsInteger;
14    dtm.cdsIntimacao.Post;
15    dtm.cdsDevedor.Next;
16   end;
17   dtm.cdsIntimacao.Next;
18  end;
19  FreeAndNil(repoDivida);
20
21  // Rotina de impressao...
22 end;
```

Listagem 3. Repositório de dividas

```
01 TDividaRepositorio = class
02 public
03   procedure LocalizarDividaPorDataEntrada(Data: TDateTime);
04 end;
05
06 //implementação
07
08 procedure TDividaRepositorio.LocalizarDividaPorDataEntrada(Data: TDateTime);
09 begin
10   dtm.cdsDivida.Close;
11   dtm.qryDivida.Close;
12   dtm.qryDivida.SQL.Clear;
13   dtm.qryDivida.SQL.Add('SELECT * FROM DIVIDA WHERE DATA_ENTRADA = ' +
14        QuotedStr(FormatDateTime('mm/dd/yyyy',Data)));
15   dtm.cdsDivida.Open;
16 end;
```

Temos a declaração do método (linha 3) e entre as linhas 8 e 16 sua implementação, que é bem simples. Contudo, mesmo simples seu valor dentro da arquitetura proposta é de extrema importância. Temos aqui a centralização de um recurso. O mesmo também pode ser visto na **Listagem 4**, que mostra o repositório de intimações.

Listagem 4. Repositório de intimações

```
01 TIntimacoesRepositorio = class
02 public
03 procedure LocalizarIntimacaoPorDevedor(IdDevedor: integer);
04 procedure LocalizarIntimacoesIntimador(IdIntimador: integer; InicioIntimacao:
   TDateTime; FimIntimacao: TDateTime);
05 procedure LocalizarIntimacaoPorId(IdIntimacao: integer);
06 procedure RegistrarRecebimento(IdIntimacao: integer; QuemRecebeu:
    string; Quando: TDateTime);
07 end;
   // implementação
08
09 procedure TIntimacoesRepositorio.LocalizarIntimacaoPorDevedo
   (IdDevedor: integer);
10 begin
11  dtm.qryIntimacao.Close;
12  dtm.qryIntimacao.SQL.Clear;
13  dtm.qryIntimacao.SQL.Add('SELECT * FROM INTIMACOES WHERE ID_DEVEDOR =
' +
14   IntToStr(IdDevedor));
15  dtm.cdsIntimacao.Open;
16 end;
17
18 procedure TIntimacoesRepositorio.LocalizarIntimacaoPorId(IdIntimacao: integer);
19 begin
20  dtm.qryIntimacao.Close;
21  dtm.qryIntimacao.SQL.Clear;
22  dtm.qryIntimacao.SQL.Add(
    'SELECT * FROM INTIMACOES WHERE ID_INTIMACAO = ' +
23   IntToStr(IdIntimacao));
24  dtm.cdsIntimacao.Open;
25 end;
26
27 procedure TIntimacoesRepositorio.LocalizarIntimacoesIntimador(
   IdIntimador: integer; InicioIntimacao, FimIntimacao: TDateTime);
28 begin
29  dtm.qryIntimacao.Close;
30  dtm.qryIntimacao.SQL.Clear;
31  dtm.qryIntimacao.SQL.Add('
    SELECT * FROM INTIMACOES WHERE ID_INTIMADOR = ' +
32   IntToStr(IdIntimador));
33  dtm.qryIntimacao.SQL.Add('AND DATA_EMISSAO BETWEEN ' +
34  QuotedStr(FormatDateTime('mm/dd/yyyy',InicioIntimacao)) + ' AND ' +
35  QuotedStr(FormatDateTime('mm/dd/yyyy',FimIntimacao)));
36  dtm.cdsIntimacao.Open;
37 end;
38
39 procedure TIntimacoesRepositorio.RegistrarRecebimento(IdIntimacao: integer;
40 QuemRecebeu: string; Quando: TDateTime);
41 begin
42  LocalizarIntimacaoPorId(IdIntimacao);
43  dtm.cdsIntimacao.Edit;
44  dtm.cdsIntimacaoRECEBIDA_EM.AsDateTime := Quando;
45  dtm.cdsIntimacaoRECEBIDA_POR.AsString := QuemRecebeu;
46  dtm.cdsIntimacao.Post;
47 end;
```

Podemos facilmente identificar na declaração da classe TIntimacoesRepositorio os métodos de negócio. É realmente muito mais entendível ler o nome do método do que identificar o que um SQL "perdido" faz dentro de uma janela. Esses repositórios continuarão acessando os componentes de dados existentes, ele está apenas criando uma camada de isolamento, que irá permitir a reutilização por outras partes do projeto.

Ao analisarmos novamente a **Figura 1**, podemos dizer que nossos repositórios e serviços estão localizados na camada de domínio, e, principalmente a camada de interface, apenas realiza um acesso a ela, sem misturar responsabilidades, como mostra a **Listagem 5**.

Listagem 5. Acessando um repositório

```
01 procedure TForm1.btnLocalizarDividasClick(Sender: TObject);
02 var
03  repo: TDividaRepositorio;
04 begin
05  repo := TDividaRepositorio.Create;
06  repo.LocalizarDividaPorDataEntrada(StrToDate(editData.Text));
07  FreeAndNil(repo);
08 end;
```

A tela principal do projeto realiza uma consulta de dívidas por data (**Figura 4**). No botão localizar é instanciado o repositório de dívidas, que já expõe um método de negócio. Ao executar o método LocalizarDividaPorDataEntrada (linha 6), a tarefa de acesso ao

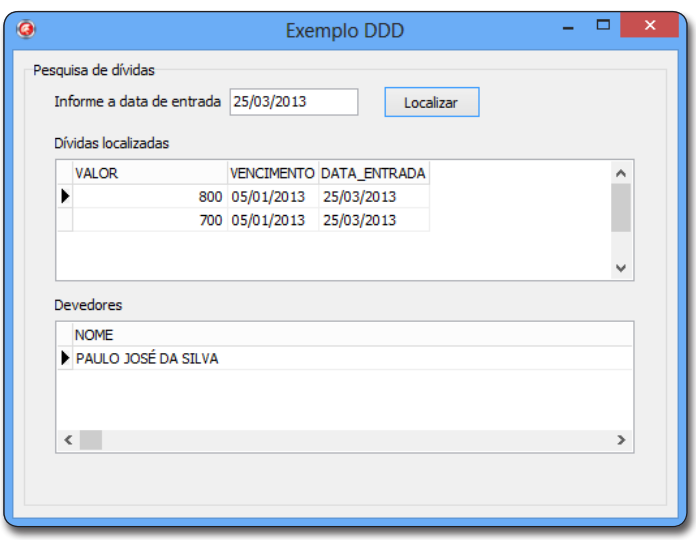

Figura 4. Tela principal

banco é delegada ao repositório, que por sua vez, fica responsável em gerar as sentenças SQL. Isso fica mais claro ao visualizar um diagrama de sequência (**Nota do DevMan 1**) dessa operação (**Figura 5**).

Com isso, temos além da centralização de acesso a dados, a organização das responsabilidades. A camada de interface, que é onde o formulário principal reside, não precisa gerar SQL algum. Caso fosse necessário expor essa consulta através da Web, apenas reutilizaríamos o repositório e serviços necessários.

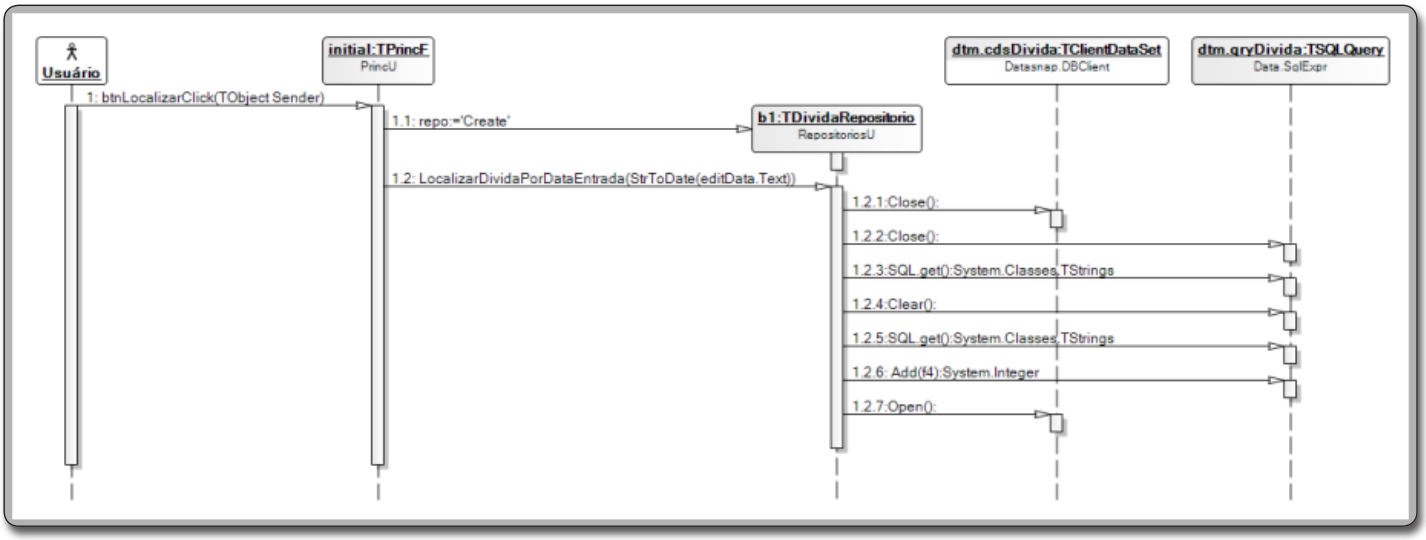

Figura 5. Diagrama de sequência

Nota do DevMan 1

Diagrama de Sequência – Enfatiza a ordem temporal de troca de mensagens entre objetos de um sistema. Por mensagens entendam-se chamadas de funções, métodos, procedimentos. Esse diagrama mostra a colaboração entre objetos em uma determinada funcionalidade do sistema, isso dá ao desenvolvedor um entendimento sobre o que acontece em determinado ponto do tempo.

É importante entender que mesmo sem aplicar a orientação a objetos em sua totalidade, podemos criar um código limpo, fácil de ser entendido e reutilizável dentro da própria aplicação.

Conclusão

O objetivo deste artigo foi apresentar o DDD e explicar suas principais ideias e padrões mais básicos, dando um exemplo de aplicação, mostrando um cenário comum ao leitor, de forma que se familiarizasse com o assunto para então buscar aplicações no seu próprio ambiente. É importante ressaltar que grande parte do que foi mostrado aqui não são conceitos novos, apenas um agrupamento de ideias já existentes.

Os métodos dos repositórios não foram implementados em sua totalidade porque o foco deveria estar no entendimento e compreensão dos problemas apresentados e da solução proposta.

É muito comum ao estudar arquitetura já buscar uma implementação pronta, mesmo sem entender sua essência. É por isso que muitos desistem dela, pois a aplicam sem entender suas motivações e consequências.

O valor de um produto não está no código final em si, mas na solução resultante do conhecimento adquirido acerca do problema.

Paulo Quicoli
pauloquicoli@gmail.com
Editor da revista ClubeDelphi e editor técnico .Net Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista programador de sistemas da Siplan Control-M (www.siplan-controlm.com.br) unidade Jaboticabal, e professor da FATEC, campus Taquaritinga. Blog: http://quicoli.worpress.com

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

LiveBindings e FireMonkey

Utilizando LiveBindings de forma simples na elaboração das aplicações

A Orientação a Objetos trouxe de fato inúmeras vantagens para a arquitetura de aplicações cada vez mais bem estruturadas, otimizadas e etc. São vários os recursos que facilitam o cotidiano da equipe de desenvolvimento, não só na elaboração, mas também durante as fases de correção e manutenção, trazendo uma diversidade de benefícios, como é o caso do aumento da produtividade e a fácil compreensão do código fonte. Até algum tempo atrás, o Delphi não trazia por padrão recursos importantes dentro deste contexto para os desenvolvedores que quisessem aplicar esta metodologia em seus softwares. Após o lançamento da versão XE 2, novas funcionalidades foram adicionadas ao RAD, possibilitando assim unir toda a interface de desenvolvimento rápido do Delphi com muitos dos recursos presentes na Orientação a Objetos, onde uma funcionalidade inserida que merece destaque é o DataBinding ou Ligação de Dados, uma tecnologia já conhecida e muito popular em outros ambientes de desenvolvimento, onde é possível efetuar a ligação de objetos à interface gráfica da aplicação de uma forma simples e rápida, em outras palavras, unir as regras de negócio a UI (User Interface). Nomeado LiveBindings na ferramenta da Embarcadero, este recurso tem por base expressões relacionais conhecidas como expressões de ligação, que podem ser unidirecionais ou bidirecionais, onde o mecanismo desta funcionalidade permite que os valores das propriedades de determinado controle final sejam alterados com base na expressão definida e um objeto de origem, sendo este recurso também disponibilizado para a ferramenta de desenvolvimento C++ Builder XE 2. Uma expressão dentro deste contexto pode ser definida como uma string que contém outra propriedade do mesmo objeto, uma propriedade de outro objeto, ou ainda ser mais complexa, possuindo operadores, funções e etc. Diante deste fator, um objeto qualquer como um botão pode ser facilmente interligado a outro objeto, como um label, uma fonte de dados ou outro botão, por exemplo, empregando tais expressões de ligação e possibilitando assim que a alteração de determinado valor do objeto em questão reflita em outro objeto, onde não são necessários códigos fontes para esta funcionalidade, salvo os casos

Resumo DevMan

LiveBindings e FireMonkey:

Com o lançamento do Delphi XE 2, os desenvolvedores puderam contar com uma diversidade de novos recursos de suma importância quando o assunto é a elaboração de aplicações melhores estruturadas, como é o caso do LiveBindings, um mecanismo que utiliza expressões para a interligação entre os objetos de uma forma rápida e simples, e o FireMonkey, a plataforma da Embarcadero que possibilita a criação de Softwares mais ricos visualmente, além de possibilitar que a aplicação em questão seja disponibilizada para vários Sistemas Operacionais. Embora sejam recursos bem recentes, tais tecnologias se popularizam cada vez mais, pois trazem uma diversidade de benefícios que podem aumentar a produtividade e possibilitam que o desenvolvedor empregue diversos conceitos de OO em seus projetos. Neste artigo, primeiramente é apresentada uma rápida introdução ao recurso de LiveBindings utilizando a plataforma FireMonkey como base, demonstrando um simples exemplo prático para conectar dois objetos através de suas propriedades, ilustrando a familiarização com tal funcionalidade. Posteriormente, é apresentado um exemplo que emprega este recurso em conjunto com DataSets para a elaboração de uma aplicação conectada a um banco de dados, onde o conceito Master-Detail é utilizado para a tabela de venda e seus itens. Para concluir, o último exemplo demonstra como empregar LiveBindings e Generics, sendo este último uma funcionalidade importante que possibilita a utilização de tipos parametrizados, com o intuito de interligar objetos criados pelo próprio desenvolvedor com o mecanismo de LiveBinding em tempo de execução, apresentando algumas das possibilidades obtidas com o emprego de tais tecnologias.

Em que situação o tema é útil:

Este artigo apresenta uma introdução ao recurso de LiveBindings e exemplos práticos de forma simplificada para o emprego desta importante funcionalidade nas aplicações cotidianas, possibilitando assim obter um aumento da produtividade no desenvolvimento de softwares para o setor comercial que empregam o conceito de Orientação a Objetos, utilizando como plataforma base o FireMonkey, mas é importante salientar que este recurso também está disponível para a plataforma VCL.

onde o mecanismo deve ser notificado caso ocorra alguma atualização das informações entre as entidades envolvidas, processo este que pode ser realizado através de eventos e métodos específicos, por exemplo. É importante ressaltar também que o recurso de LiveBindings não só pode ser empregado para os objetos adicionados em tempo de Design, mas também para os criados em tempo de execução, onde simples trechos de código podem ser utilizados, unindo recursos interessantes como Generics, que em suma são tipos parametrizados. Para empregar o recurso de LiveBindings em seus projetos, o desenvolvedor conta com diversas alternativas, podendo utilizar da propriedade LiveBinding presente em determinados controles, a opção New LiveBinding presente no menu de contexto dos objetos suportados, inserir os componentes manualmente, dentre outras formas.

Quanto à estrutura do mecanismo de LiveBindings, ela é constituída basicamente pela origem, expressão de origem, controle e expressão de controle. A origem pode ser um elemento qualquer como um objeto, um DataSet, uma lista e etc. A expressão de origem é uma string onde o nome de determinada propriedade do objeto de origem é informado para que a ligação seja realizada, podendo também conter uma expressão melhor elaborada, contendo uma função, tal como UpperCase(Text). O controle é o objeto na qual determinada informação será visualizada, em outras palavras, o destino final da mesma. Quanto à expressão de controle, ela também é uma string, porém, deve conter a propriedade alvo do objeto de controle que será afetada. A **Figura 1** apresenta um simples exemplo da estrutura básica do recurso de LiveBindings.

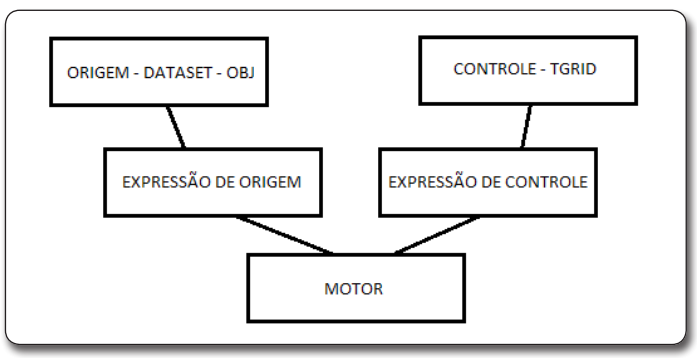

Figura 1. Exemplo da estrutura básica de LiveBindings

Outro recurso de bastante notoriedade e que tem gerado diversos adeptos (também incluído a partir da versão XE 2) é o FireMonkey, conhecido também pela sigla FMX. Consiste em uma plataforma para o desenvolvimento de aplicações visualmente mais ricas e refinadas, possibilitando que o software seja disponibilizado para outros Sistemas Operacionais além do Windows, como é o caso do Mac OS, empregando o conceito de elaborar a aplicação apenas uma vez e assim disponibilizá-la para outros ambientes Sem dúvida uma funcionalidade almejada por qualquer equipe de desenvolvimento. No FireMonkey, a interface gráfica do software projetado pode ser muito mais interativa com o usuário final, proporcionando assim uma experiência mais agradável para o mesmo, unindo animações, recursos 2D/3D e diversas funcionalidades presentes na atualidade. É importante ressaltar também que esta plataforma utiliza os recursos nativos da CPU (Unidade Central de Processamento) e da GPU (Unidade de Processamento Gráfico), permitindo assim que tais aplicações tenham seu desempenho otimizado, apresentando diversas vantagens com relação a softwares que necessitam de máquinas virtuais, dentre outros recursos.

O FireMonkey possui seu próprio motor GUI (Interface Gráfica do Usuário), onde componentes como botões, janelas, caixas de texto e etc, são independentes do Sistema Operacional em questão. Quanto às animações, elas são realizadas em segundo plano através de uma Thread (**Nota do DevMan 1**), utilizando poucos recursos da CPU.

Nota do DevMan 1

Thread é a unidade básica relacionada a uma aplicação em execução nos Sistemas Operacionais tradicionais como Windows e Unix Like. Em softwares Multithread, este conceito é muito mais explorado, onde diversas técnicas e recursos da atualidade possibilitam que Threads sejam executadas de forma simultânea, onde tarefas podem ocorrer paralelamente e as mesmas trabalhem de maneira independente umas das outras. Também é importante ressaltar que as Threads de determinado processo compartilham o mesmo intervalo de endereço, o que possibilita a fácil troca de informações entre ambas. Outro ponto importante dentro deste contexto é o escalonamento (scheduling), que por sua vez é uma atividade realizada para possibilitar que vários processos executem em paralelo, tendo como objetivo o aumento da produtividade, uma vez que o processador é melhor aproveitado e são minimizados o tempo de resposta ao usuário final e a sobrecarga de recursos, contando ainda com mecanismos que possibilitam a atribuição de prioridades para determinados processos.

Com tantos benefícios, empregar o recurso de LiveBinding nas aplicações é sem dúvida um fator importante, pois como mencionado, apresenta uma nova forma de efetuar ligações entre as entidades envolvidas no software, e indo mais além, possibilita que diversos conceitos da Orientação a Objetos possam ser empregados sem a necessidade de códigos redundantes. Com relação ao FireMonkey, também é uma tecnologia que merece destaque, afinal o fato de desenvolver uma aplicação escrita uma vez e compilada para mais de um Sistema Operacional é uma vantagem muito significativa, além evidentemente, de todos os recursos gráficos obtidos ao utilizá-la.

Componentes importantes para o emprego de LiveBindings

O Delphi XE 2 conta com alguns novos componentes para o emprego do recurso de LiveBindings, presentes na guia de mesmo nome, bem como uma diversidade de classes que podem ser instanciadas, presentes em Units como Data.Binding.Components, System.Bindings.Outputs, dentre outras. A **Tabela 1** apresenta tais componentes e uma breve descrição sobre os mesmos.

Entretanto, se o desenvolvedor utilizar um dos assistentes ou o menu de contexto para implementar os recursos de LiveBindings, o Delphi criará automaticamente os componentes necessários e efetuará a devida configuração dos mesmos, restando ao desenvolvedor apenas efetuar algum ajuste adicional se assim necessitar.

Componente	Descrição
TBindSourceDB	Um dos componentes básicos para o emprego do recurso LiveBindings cujo objetivo é possibilitar a interligação com uma fonte de dados, como um DataSet.
TBindScope	Este também é uma das bases para a utilização de LiveBindings, sendo responsável por prover a ligação entre os componentes.
TBindingsList	Tem como objetivo principal, a elaboração de listas de ligação, local onde os vínculos estão definidos.
TBindNavigator	Possibilita a navegação entre os registros de determinada fonte de dados, se assemelhando ao tradicional componente TDBNavigator.

Tabela 1. Componentes básicos para o emprego do recurso de LiveBindings no Delphi XE 2

Criando uma simples aplicação empregando Live Bindings em controles

Para se familiarizar com este recurso, neste primeiro exemplo esta funcionalidade será empregada de uma forma extremamente simples, unindo dois componentes, cujo intuito será aumentar ou diminuir a fonte de um texto presente em um objeto TLabel através de um objeto TTrackBar. Sendo assim, um novo projeto de aplicação FireMonkey deve ser iniciado (File>New>FireMonkey HD Application). No formulário principal, deve ser adicionado um objeto do tipo TLabel e um do tipo TTrackBar. Na propriedade Text do Label1 foi adicionado o texto "Live Bindings" e definido como negrito e tamanho 18. Quanto ao objeto TrackBar1, algumas de suas propriedades receberam as definições de acordo com o trecho da **Listagem 1**.

Lembrando que a propriedade Max é utilizada para definir o valor máximo para o TrackBar, Min o valor mínimo, e Value o valor inicial. Definidas estas propriedades, o próximo passo é selecionar o objeto Label1 e através de seu menu de contexto, acessar a opção New LiveBinding para que a janela de criação do mecanismo de LiveBindings seja exibida e, por meio dela, um novo objeto TBindExpression seja criado, sendo que esta classe possibilita a implementação de um tipo de expressão de ligação. A **Figura 2** apresenta a janela New LiveBinding.

Listagem 1. Definições do objeto TrackBar1

```
01 Max = 72
02 Min = 8
03 Value = 18
```

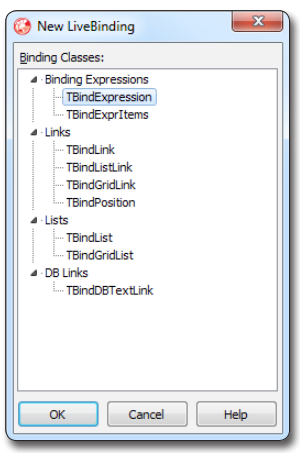

Figura 2. Janela New LiveBinding

Neste caso, como a janela New LiveBinding foi utilizada, o Delphi cria automaticamente um novo objeto TBindingsList e uma nova expressão (TBindExpression). Vale a pena ressaltar que para realizar a configuração das expressões, basta efetuar um clique sobre o objeto TBindingsList. Com a expressão adicionada selecionada, devem ser definidas algumas de suas propriedades através do Object Inspector de acordo com a **Listagem 2**.

Listagem 2. Propriedades da expressão selecionada

```
01 ControlExpression = Font.Size
02 SourceComponent = TrackBar1
03 SourceExpression = Value
```

Através destas propriedades, foram definidas a expressão de controle com o valor Font.Size (linha 01), que neste caso, equivale a propriedade do tamanho da fonte do objeto Label1, o objeto de origem (linha 02) foi definido como TrackBar1 e por último, a expressão de origem (linha 03), recebeu a propriedade Value do objeto TrackBar1, indicando que a mesma deve ser utilizada para efetuar a ligação com a propriedade Font.Size.

Neste momento a configuração das propriedades dos componentes está concluída, entretanto, cada vez que o objeto TrackBar1 tiver seu valor alterado o mecanismo de LiveBinding precisa ser notificado que houve uma atualização. Sendo assim, uma forma de realizar este processo é através do evento OnChange do objeto de Origem (TrackBar1), onde o mesmo deve receber o simples trecho de código da **Listagem 3**.

Listagem 3. Evento OnChange do TrackBar1

```
01 BindingsList1.Notify(Sender,'Value');
```

Com o emprego do método Notify (linha 01), o mecanismo de LiveBindings é notificado sobre a alteração na devida propriedade e realiza a atualização. O primeiro parâmetro deste método é objeto que sofreu a alteração, neste caso o próprio TrackBar1 (Sender), e o segundo, a propriedade afetada (Value), que todavia também poderia ser informada em branco ('') se houvessem outras propriedades envolvidas, ocasionando uma notificação geral.

Neste momento, a aplicação está finalizada e já pode ser executada. Este simples exemplo demonstra a facilidade de empregar LiveBindings e interligar um objeto a outro de uma forma rápida. A **Figura 3** apresenta o visual final desta aplicação.

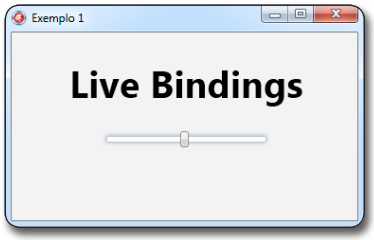

Figura 3. Aplicação do Exemplo 1 em execução

Criando o banco de dados que será utilizado no Exemplo 2

Antes de dar início a outro exemplo prático empregando o recurso de LiveBindings com DatasSets, será criado um novo banco de dados utilizando o SGBD Firebird, onde o mesmo recebeu a denominação de LIVEBINDINGS.FDB. Neste banco são criadas duas tabelas, uma de Vendas e Itens da Venda respectivamente, para que futuramente o conceito de Master-Detail seja empregado na aplicação. A **Listagem 4** apresenta o SQL utilizado na elaboração da tabela principal, VENDA.

No trecho da **Listagem 4** é possível notar que a tabela conta com os campos mais comuns em uma venda, tendo como chave primária o campo VENDA_CODIGIO. A **Listagem 5** ilustra o SQL utilizado para a criação da tabela de Itens da Venda (VENDA_ITENS).

Listagem 4. SQL para a crição da tabela Venda

```
CREATE TABLE VENDA(
 VENDA_CODIGO INTEGER,
 CLIENTE_CODIGO INTEGER,
 VENDA_DATA TIMESTAMP,
 VENDA_PGTO VARCHAR(5),
 VENDA_TOTAL NUMERIC(9,2),
 VENDA_STATUS VARCHAR(1),
 CONSTRAINT PKY_VENDA PRIMARY KEY(VENDA_CODIGO)
);
```

Listagem 5. SQL para a crição da tabela de Itens da Venda

```
CREATE TABLE VENDA_ITENS(
 VENDA_CODIGO INTEGER,
 VENDA_ITENS_ITEM INTEGER,
 VENDA_ITENS_DESCRICAO VARCHAR(50),
 VENDA_ITENS_QUANTIDADE NUMERIC(9,3),
 VENDA_ITENS_VALOR NUMERIC(9,2),
 VENDA_ITENS_TOTAL NUMERIC(9,2),
 CONSTRAINT PKY_VENDA_ITENS PRIMARY KEY(VENDA_CODIGO,
  VENDA_  ITENS_ITEM),
 CONSTRAINT FKY_VENDA_ITENS FOREIGN KEY(VENDA_CODIGO) REFERENCES
  VENDA(VENDA_CODIGO) ON UPDATE CASCADE ON DELETE CASCADE
);
```

No trecho da **Listagem 5** é possível notar que a tabela de itens da venda possui chave primária composta, onde é utilizado o código da venda e o número do item para a mesma, uma vez que o campo VENDA_CODIGO será utilizado posteriormente para o mestre-detalhe.

Criando a aplicação e configurando a conexão com o banco de dados

Para dar início a um exemplo simples empregando LiveBindings com DataSets, um novo projeto FireMonkey deve ser iniciado. Para a conexão com o banco de dados, o framework DBExpress será utilizado, portanto, um objeto do tipo TSQLConnection deve ser inserido no formulário principal. Vale à pena lembrar que este objeto é o responsável por efetuar a conexão com o servidor de banco de dados. Sendo assim, ele deve ser configurado de acordo com o trecho da **Listagem 6.**

Listagem 6. Configurações do SQLConnection

```
01 ConnectionName = FBConnection
02 Params.Database = LIVEBINDINGS.FDB
03 LoginPrompt = False
04 Connected = True
```

Através da propriedade ConnectionName (linha 01), é definido o tipo da conexão, Firebird (FBConnection). Em Params.Database (linha 02) é definido o endereço da base de dados. Também é setada a propriedade LoginPrompt (linha 03) com o valor False, fazendo com que a tela de Login de usuário do banco de dados não seja exibida, uma vez que as informações de usuário já estão definidas em Params.User_Name e Params.Password, e para concluir, a conexão é ativada através de Connected := True (linha 04).

O próximo passo é adicionar ao formulário principal, dois objetos do tipo TSimpleDataSet. Este objeto pertencente ao framework DBXpress, é responsável por efetuar um cache das informações na memória de acordo com o comando SQL empregado e possibilitar a realização das operações básicas para os dados (CRUD – Create, Read, Update and Delete). Sendo assim, os mesmos foram nomeados sdsVenda e sdsVendaItens, com suas propriedades Connection apontadas para o componente SQLConnection1. Em seguida, a propriedade DataSet.CommandText de ambos os DataSets deve ser definida, lembrando que é através dela que a instrução SQL desejada é informada. Logo tal propriedade dos componentes sdsVenda e sdsVendaItens receberam respectivamente a instrução informada na **Listagem 7**.

Listagem 7. Instruções SQL dos componentes sdsVenda e sdsVendaItens

```
[sdsVenda]
SELECT * FROM VENDA
[sdsVendaItens]
SELECT * FROM VENDA_ITENS
```

Após a definição dos comandos, é interessante adicionar os TFields para os DataSets através de um clique duplo sob os mesmos e selecionando de direita no Fields Editor a opção "Add All Fields" do menu de contexto. Para possibilitar que as alterações realizadas nos registro sejam efetivadas no banco de dados, os eventos AfterDelete e AfterPost de ambos devem receber o método ApplyUpdates, tal como a **Listagem 8.**

Listagem 8. Codificação atribuída aos eventos AfterDelete e AfterPost dos SimpleDataSets

```
01 sdsVenda.ApplyUpdates(0);
```

Definidos os códigos para os eventos de ambos os DataSets, conforme a **Listagem 8**, devem ser adicionados dois novos objetos do tipo TDataSource ao formulário, sendo os mesmos nomeados para dsVenda e dsVendaItens. Lembrando que este componente possibilita a interligação dos componentes visuais às fontes de dados. Ambos os DataSources devem ser interligados aos DataSets através da propriedade DataSet. Para concluir esta etapa, a configuração para o emprego do conceito Master-Detail deve ser realizada. Sendo assim, a propriedade MasterSource do objeto sdsVendaItens recebe o valor dsVenda, realizando a ligação com o mesmo. Para indicar os campos que possuem o relacionamento, a janela Field Link Designer deve ser utilizada através da propriedade MasterFields, onde o campo VENDA_CODIGO deve ser selecionado em ambos os recipientes (Details Fields e Master Fields) e adicionado através do botão Add. Neste momento, os DataSets estão devidamente configurados e relacionados para o Master-Detail.

Empregando o recurso de LiveBindings no Exemplo 2

Como visto até o momento, a utilização do recurso de LiveBindings é um processo simples e rápido que pode ser realizado de várias formas. Neste segundo exemplo, antes de iniciar a implementação do mecanismo, será adicionado um objeto do tipo TTabControl ao formulário, onde posteriormente devem ser inseridas duas abas através da janela Items Editor. Na primeira aba, serão adicionados os controles visuais para o preenchimento das informações da Venda, e na segunda, os Grids para a visualização completa dos dados. Sendo assim, na segunda aba foram inseridos dois objetos do tipo TStringGrid. Com o primeiro TStringGrid selecionado, para iniciar o uso do LiveBinding, basta selecionar a opção Link to DB DataSource, presente na propriedade LiveBindings. A **Figura 4** apresenta as opções para a definição do recurso de LiveBindings através das propriedades dos componentes.

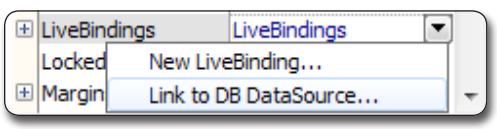

Figura 4. Opções da propriedade LiveBindings

Ao utilizar tal opção, a janela New DB Link é aberta e apresenta os DataSets disponíveis para que a ligação seja realizada. Dando sequência ao projeto, basta selecionar o objeto dsVenda para o StringGrid1. Concluída esta etapa, é possível notar que são adicionados automaticamente, dois novos objetos ao formulário, sendo eles o TBindScopeDB e o TBindingList, lembrando que o primeiro é utilizado para a ligação com o DataSet e o segundo para a definição da lista contendo as expressões. Para o objeto StringGrid2, o mesmo processo deve ser realizado, porém, selecionando o objeto dsVendaItens. É possível notar que um novo objeto TBingScopeDB é criado e que uma nova expressão é adicionada no objeto BindingList1. Para efetuar a navegação entre os registros e as operações CRUD, um componente do tipo TBindNavigator deve ser inserido no formulário principal. Como se percebe, este objeto se assemelha ao tradicional componente DBNavigator. Para utilizar seus recursos, basta definir sua propriedade BindScope com o valor BindScopeDB1.

O próximo passo para este exemplo é a adição de quatro componentes do tipo TEdit e dois do tipo TComboBox. Para empregar o recurso de LiveBindings a eles, também é utilizada a mesma propriedade utilizada para os objetos TStringGrid (LiveBindings), bem como a opção Link to DB Field, onde serão exibidos todos os campos dos DataSets, presentes nos objetos BindScopeDB1 e BindScopeDB2, restando apenas informar a qual destes o controle atual será interligado. A **Figura 5** demonstra a janela New DB Link apresentando os campos já disponibilizados.

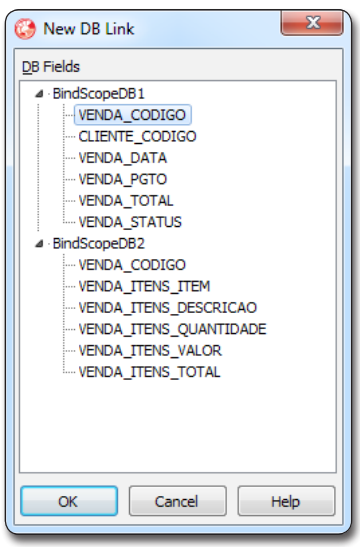

Figura 5. Janela New DB Link com os campos já disponibilizados

Entretanto, existe outra maneira de realizar a interligação dos componentes visuais com seus devidos campos, criando manualmente uma nova expressão, onde tal processo pode ser realizado através de um clique duplo sob o objeto BindingsList1, conforme mencionado anteriormente. Sendo assim, a janela de edição da lista de expressões de ligação será aberta, possibilitando a alteração, exclusão ou adição de novas expressões. A **Figura 6** demonstra a janela para a edição da lista de expressões.

Sendo assim, para inserir uma nova expressão manualmente, basta utilizar o botão New Binding. Neste exemplo, esta alternativa será empregada para criar uma expressão de forma manual para o objeto TEdit referente ao campo data. Com a janela New LiveBinding aberta, a opção TBindDBEditLink deve ser selecionada. A **Figura 7** apresenta a janela New LiveBinidng com a opção TBindDBEditLink em evidência.

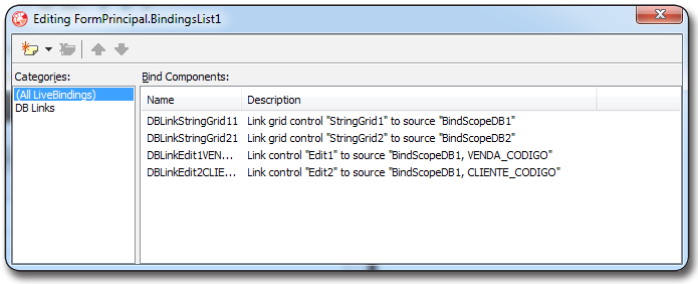

Figura 6. Janela de edição da lista de expressões

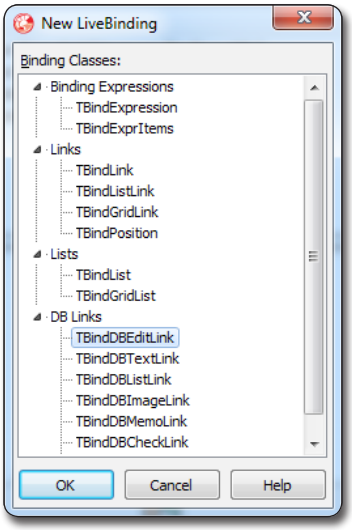

Figura 7. Janela New LiveBinding

Ao realizar este processo, uma nova expressão é criada. O próximo passo é efetuar a edição da mesma. Inicialmente, a propriedade DataSource deve ser definida com o valor de BindScopeDB1. Em seguida, basta definir EditControl com o devido objeto TEdit que exibirá a informação, e para finalizar, o nome do campo (VENDA_DATA), semelhante ao que fazemos com controles Data Aware

Com relação aos objetos do tipo TComboBox, para a forma de pagamento e status da venda, o processo de LiveBindings é realizado da mesma forma, através da opção Link To DB Field e da definição do campo VENDA_PGTO ou VENDA_STATUS, ou ainda através do menu de contexto.

Neste momento a aplicação de exemplo está concluída e já pode ser executada. É importante verificar se ambos os DataSets estão com a propriedade Active definida como True para a abertura automática. A **Figura 8** demonstra o visual final da aplicação do Exemplo 2.

Criando uma classe para o uso de LiveBindings e Generics

Outra possibilidade muito interessante ao utilizar LiveBindings é empregar o recurso de Generics. Este foi incorporado ao Delphi a partir da versão 2009 e trouxe diversos benefícios. De forma resumida, esta funcionalidade possibilita que tipos parametrizados possam ser empregados na definição de métodos, classes, registros e interfaces com uma maior flexibilidade. Ele se torna um grande aliado quando o assunto é criar listas compostas por quaisquer objetos, inclusive os instanciados de classes definidas pelo próprio

desenvolvedor, o que permite uma maior organização e facilidade para acesso, como é o caso do emprego da classe TList para obter esta funcionalidade. Quanto ao emprego de Generics em métodos, existe a possibilidade de um objeto genérico definido ser resolvido em tempo de execução, sendo posteriormente substituído por um tipo, de forma semelhante ao processo que ocorre nas funções e procedimentos, onde os valores são informados através dos argumentos, porém nestes últimos não há a mesma flexibilidade, uma vez que os tipos para os mesmos devem ser pré-definidos.

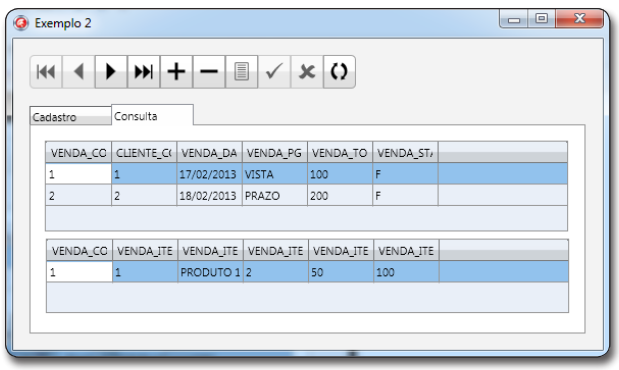

Figura 8. Aplicação com LiveBindings, DataSets e conexão com banco de dados

Para dar início a um exemplo prático do emprego destes recursos, um novo projeto FireMonkey deve ser criado. Neste terceiro exemplo, o primeiro passo é criar a classe que dará origem aos objetos. Sendo assim, será criada uma simples classe baseada em Plain Old Delphi Object para possibilitar o cadastro de carros e algumas de suas características, simulando o processo que ocorre no mundo real, onde a mesma recebeu a denominação de TCarro. A **Listagem 9** demonstra a estrutura da classe.

Como é possível notar, esta classe possui quatro atributos, onde são definidos através de seus devidos métodos set e podem ser acessados por meio de suas propriedades. Vale à pena lembrar que os métodos do tipo set podem ser gerados automaticamente pelo Delphi através das teclas CTRL + SHIFT + C e são constituídos pela estrutura FAtributo := Value.

Definida a classe TCarro, a próxima etapa deste projeto é declarar algumas variáveis globais para o formulário principal. Sendo assim, sua seção var deve receber a definição do código da **Listagem 10**.

Como pode ser observado na **Listagem 10**, a variável Carros foi declarada como uma lista de objetos (TList) do tipo TCarro (linha 02), empregando o recurso de Generics, lembrando que a Unit System.Generics.Collections deve ser adicionada à sessão Uses do formulário. A variável BindSope foi declarada como TBindScope (linha 03), o objeto que possibilita a ligação entre os componentes, onde a Unit Data.Binding.Components também deve ser incluída na cláusula Uses. Já a variável BindGridList foi declarada como TBindGridList (linha 04), que por sua vez deve ser empregada para efetuar a ligação do objeto com um Grid, como é o caso do TStringGrid. A variável ExpressaoColuna foi declarada como TColumnFormatExpressionItem (linha 05), sendo a função deste último descrever a expressão string e o elemento que determinado componente irá empregar para a criação de expressões de ligação formatadas para uma coluna do Grid. Para concluir, foi declarada a variável ExpressaoCelula, do tipo TExpressionItem (linha 06), sendo este tipo utilizado para a definição de um item em uma coleção de expressões.

O próximo passo no projeto é inicializar a variável denominada Carros (TList) em conjunto com o formulário principal da aplicação. Portanto, o evento OnCreate do mesmo deve receber o trecho de código da **Listagem 11**.

Listagem 9. Estrutura da classe TCarro

```
01 type
02 TCarro = class
03 private
04  FModelo: string;
05  FMarca: string;
06  FCor: string;
07  FProprietario: string;
08  procedure SetModelo(const Value: string);
09  procedure SetMarca(const Value: string);
10  procedure SetCor(const Value: string);
11  procedure SetProprietario(const Value: string);
12 published
13 public
14  property Modelo: string read FModelo write SetModelo;
15  property Marca: string read FMarca write SetMarca;
16  property Cor: string read FCor write SetCor;
17  property Proprietario: string read FProprietario write SetProprietario;
18 end;
```

Listagem 10. Criação das variáveis globais

```
01 var
02 Carros = TList<TCarro>;
03 BindScope: TBindScope;
04 BindGridList: TBindGridList;
05 ExpressaoColuna: TColumnFormatExpressionItem;
06 ExpressaoCelula: TExpressionItem;
```

Listagem 11. Criação da lista de TCarro no evento OnCreate do formulário

```
01 Carros := TList<TCarro>.Create;
```

Para efetuar a criação dos objetos, o carregamento das informações e realizar o processo de LiveBindings com Generics, foi adicionado ao formulário um novo botão (TButton). Este botão recebeu o nome de btCarregarDados. Também deve ser inserido um objeto do tipo TStringGrid para que as informações dos carros sejam exibidas. A **Listagem 12** apresenta o código que deve ser definido no evento OnClick do mesmo.

Inicialmente, nas linhas 02 e 03 é definida uma constante contendo cada propriedade da classe TCarro para futuramente preencher os títulos das colunas do objeto StringGrid1. Em seguida, entre as linhas 04 e 06, é declarada uma variável Carro para possibilitar a criação de alguns objetos do tipo carro para o exemplo, e duas do tipo inteiro para auxiliar no futuro emprego dos laços while e for (mais adiante). Na linha 08, é criado o primeiro objeto instanciado da classe TCarro através do método Create. Posteriormente, entre as linhas 09 e 15, são definidas preenchidas as propriedades para o

Listagem 12. Código do Evento OnCreate do botão btCarregarDados

```
01 procedure TForm1.btCarregarDadosClick(Sender: TObject);
02 const
03  ArrayTitulos: array[0..3] of string = ('Modelo','Marca','Cor','Proprietário');
04 var
05  Carro: TCarro;
06  NumeroCarros, i: Integer;
07 begin
08  Carro := TCarro.Create;
09  with Carro do
10  begin
11   SetModelo('Corsa');
12   SetMarca('GM');
13   SetCor('Prata');
14   SetProprietario('João');
15  end;
16  Carros.Add(Carro);
17  Carro := TCarro.Create;
18  with Carro do
19  begin
20   SetModelo('Gol');
21   SetMarca('VW');
22   SetCor('Branco');
23   SetProprietario('Maria');
24  end;
25  Carros.Add(Carro);
26  Carro := TCarro.Create;
27  with Carro do
28  begin
29   SetModelo('Civic');
30   SetMarca('Honda');
31   SetCor('Preto');
32   SetProprietario('Marcos');
33  end;
34  Carros.Add(Carro);
35  Carro := TCarro.Create;
36  with Carro do
37  begin
38   SetModelo('Pálio');
39   SetMarca('Fiat');
40   SetCor('Vinho');
41   SetProprietario('Márcia');
42  end;
43  Carros.Add(Carro);
44  NumeroCarros := Carros.Count;
45  while StringGrid1.ColumnCount < 4 do
46   StringGrid1.AddObject(TStringColumn.Create(Self));
47  for i := 0 to StringGrid1.ColumnCount - 1 do
48   StringGrid1.Columns[i].Header := ArrayTitulos[i];
49  BindScope := TBindScope.Create(Self);
50  BindGridList := TBindGridList.Create(Self);
51  BindGridList.ControlComponent := StringGrid1;
52  BindGridList.SourceComponent := BindScope;
53  ExpressaoColuna := BindGridList.ColumnExpressions.AddExpression;
54  ExpressaoCelula := ExpressaoColuna.FormatCellExpressions.AddExpression;
55  ExpressaoCelula.ControlExpression := 'cells[0]';
56  ExpressaoCelula.SourceExpression := 'current.Modelo';
57  ExpressaoColuna := BindGridList.ColumnExpressions.AddExpression;
58  ExpressaoCelula := ExpressaoColuna.FormatCellExpressions.AddExpression;
59  ExpressaoCelula.ControlExpression := 'cells[1]';
60  ExpressaoCelula.SourceExpression := 'current.Marca';
61  ExpressaoColuna := BindGridList.ColumnExpressions.AddExpression;
62  ExpressaoCelula := ExpressaoColuna.FormatCellExpressions.AddExpression;
63  ExpressaoCelula.ControlExpression := 'cells[2]';
64  ExpressaoCelula.SourceExpression := 'current.Cor';
65  ExpressaoColuna := BindGridList.ColumnExpressions.AddExpression;
66  ExpressaoCelula := ExpressaoColuna.FormatCellExpressions.AddExpression;
67  ExpressaoCelula.ControlExpression := 'cells[3]';
68  ExpressaoCelula.SourceExpression := 'current.Proprietario';
69  BindScope.DataObject := Carros;
70 end;
```

novo objeto. Na linha 16, o objeto carro é adicionado à lista genérica criada anteriormente (Carros do tipo TList). Da linha 17 até a linha 43, tais processos são novamente realizados para a criação de três novos carros, e consequentemente, a adição dos mesmos à lista Carros. Na linha 44, a variável NumeroCarros recebe o total de objetos presentes na lista através da propriedade Count. Em seguida, nas linhas 45 e 46 é realizado um laço while para a criação de quatro colunas no objeto StringGrid1, uma vez que as mesmas serão utilizadas para a exibição das informações dos objetos, como já mencionado. Entre as linhas 47 e 48, é efetuado outro laço, agora do tipo for, com o objetivo de percorrer todas as colunas criadas e atribuir seus respectivos títulos através da propriedade Header e com o auxílio do array criado inicialmente (ArrayTitulos). Na linha 49, o objeto BindScope, declarado anteriormente na seção var do formulário, é devidamente criado, assim como o objeto BindGridList na linha 50. Na linha 51 é definido o objeto de controle (destino) para o BindGridList, o StringGrid1, e na linha 52, o objeto de origem, BindScope. Na linha 53, é criada uma nova expressão de coluna para o objeto BindGridList, sendo a mesma atribuída a variável ExpressaoColuna. Posteriormente (linha 54), é criada uma nova expressão de célula para o objeto ExpressaoColuna, sendo o mesmo atribuído a variável ExpressaoCelula. Para definir a expressão de controle cujo objetivo final será a primeira célula do Grid (linha 55), a propriedade ControlExpression do objeto ExpressaoColuna é definido com o valor cells[0]. Quanto a expressão de origem (linha 56), é definido o valor current.Modelo, lembrando que tal informação está vinculada ao objeto BindScope e, consequentemente, a lista (Carros), então é utilizado o objeto atual (current) e sua propriedade que deve ser lida, neste caso, Modelo. Da linha 57 até a linha 68, o mesmo processo é realizado para que a ligação entre as outras células e as propriedades restantes dos objetos do tipo TCarro sejam efetuadas. Para concluir, na linha 69 a ligação entre o objeto BindScope e a lista contendo os carros é realizada, criando assim o devido vínculo entre ambos.

Neste momento a aplicação já pode ser executada. A **Figura 9** demonstra o visual final da aplicação.

Com a aplicação em execução e após efetuar um clique no botão Carregar Dados, os objetos são devidamente criados e as informações exibidas no controle rapidamente, apenas utilizando o mecanismo de LiveBindings. Neste exemplo é possível notar um grande benefício deste recurso, a ligação direta entre objetos elaborados pelo próprio desenvolvedor com o controle TStringGrid, lembrando que esta última funcionalidade também pode ser empregada facilmente utilizando outros componentes como TEdits, TComboBox e etc.

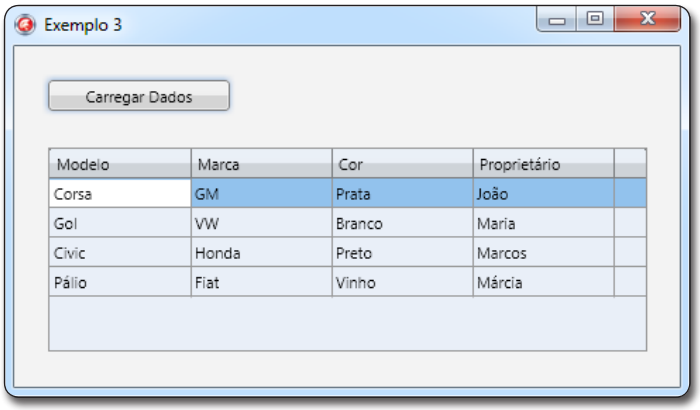

Figura 9. Aplicação com LiveBindings e Generics em execução

Conclusão

A partir do lançamento da versão XE 2, muitos recursos interessantes foram introduzidos ao Delphi, fazendo com que a ferramenta já famosa pelo desenvolvimento rápido de aplicações, disponibilizasse ainda mais funcionalidades significativas e se tornasse mais competitiva em relação a outras da atualidade. O recurso de LiveBindings e o FireMonkey, duas das principais novidades introduzidas nesta versão, trouxeram ao Delphi novas formas de projetar e criar softwares cada vez mais robustos e otimizados. Como visto durante os exemplos deste artigo, o mecanismo de LiveBindings é muito simples e rápido de ser empregado nos mais diversos projetos, na qual bastam apenas alguns cliques e definições de propriedades para que o mesmo possa ser utilizado, e em alguns casos, curtos trechos de código para a realização das notificações. Além do mais, unindo LiveBindings a recursos como Generics, ou ainda, ferramentas de ORM (Mapeamento Objeto Relacional), como é o caso do DORM e do Aurelius, o desenvolvedor pode elaborar aplicações utilizando as melhores práticas da Orientação a Objetos, conseguindo separar as regras de negócio da interface de usuário e assim, obter os diversos benefícios que esta metodologia oferece. É importante ressaltar também que o recurso de LiveBindings está disponível não só para a plataforma FireMonkey, mas também para a VCL.

De fato, o LiveBindings é uma alternativa aos tradicionais componentes Data Aware e sem dúvida, uma funcionalidade muito mais abrangente que realmente vale a pena ser estudada, compreendida e aplicada nos projetos do dia-a-dia.

Giuliano Scombatti Pinto
giuliano@sygnux.com.br
É Analista Desenvolvedor da Sygnux Software (www.sygnux.com.br) localizada em Monte Alto/SP, atuando também com desenvolvimento Web e Android. Formado pela Fatec/TQ, começou a desenvolver com Delphi em 2003. Atualmente trabalha com Delphi, Java, JavaScript, PHP e Flash/Flex. Professor de Informática de Projetos Sociais da Prefeitura Municipal de Monte Alto/SP.

Maiores informações sobre LiveBindings
http://docwiki.embarcadero.com/RADStudio/XE2/en/LiveBindings_in_RAD_Studio

DevSpace do Autor
http://www.devmedia.com.br/space.asp?id=283274

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Pronto para Web com DataSnap XE3

Conheça um pouco sobre a tecnologia mais poderosa do Delphi

Embora o Delphi tenha ficado famoso pela sua capacidade RAD de criar aplicações desktop de alto desempenho e com bastante facilidade, não é de hoje que se sabe que ele também é uma poderosa ferramenta para o desenvolvimento de aplicações web completas. O Delphi possibilita diversas formas de criação de aplicações web dinâmicas, através de várias tecnologias, como:

WebBroker: é um modelo de desenvolvimento baseado em WebActions. Para os que usam Servlets e agora o ASP.NET MVC, este não é um termo novo. É um dos primeiros modelos de desenvolvimento web disponíveis no Delphi que respondem a estímulos de Response e Request. É de fácil de aprendizado, mas requer conhecimento em HTML;

WebSnap: similar ao WebBroker que introduz conceitos como Server Side Script, inexistente no outro. Esse modelo serviu de base para os controles de servidor que temos no IntraWeb. Embora o WebSnap ainda esteja disponível no Delphi, ele foi sinalizando em algumas versões como "deprecated" (não mais atualizado). Como alternativa, comece a usar o IntraWeb (VCL for the Web);

IntraWeb (VCL for the Web): surgiu na versão 7 do Delphi trazendo uma metodologia RAD para o desenvolvimento de aplicações Web. Montado sobre a estrutura do WebSnap, possibilitou que programadores de aplicações Desktop com Delphi se sentissem facilmente adaptados ao modelo web, por tornar o desenvolvimento dessas aplicações antes baseado em um modelo de formulários;

Delphi Prism ASP.NET: ganhou vários adeptos dentro do modelo de desenvolvimento web usando o .NET framework. Uma evolução do modelo ASP, seu antecessor, baseado em script de servidor da Microsoft – uma estrutura de páginas HTML e uma estrutura de script code behind para tornar a aplicação dinâmica.

Resumo DevMan

Pronto para Web com DataSnap XE3:

É muito comum a situação onde um cliente solicita que alguma informação de seu sistema esteja disponível online, na internet. Muitos desenvolvedores partem para utilização de outras ferramentas, esquecendo que o Delphi oferece suporte ao desenvolvimento de aplicações Web. Nesse artigo serão abordadaos, de forma sucinta, os recursos oferecidos pelo DataSnap XE3 e que muitas vezes passam despercebidos pelo desenvolvedor na correria de seu dia a dia.

Em que situação o tema é útil:

Aproveitar o conhecimento já existente na ferramenta e criar extensões para WEB, de seu aplicativo, de uma rotina específica, possibilitando a integração de negócios.

Entretanto, alguns desenvolvedores Delphi ainda não se atentaram ao poder que a nova estrutura do DataSnap tem para o desenvolvimento de aplicações web de alta performance. É sobre esta abordagem de desenvolvimento web que trataremos neste artigo.

REST e JSON

Representational State Transfer (REST) é uma arquitetura para Web Services. Grandes fornecedores como Google, Yahoo, Amazon e Microsoft, têm aderido a este modelo para a maioria dos seus novos serviços web, devido ao seu ótimo desempenho, além da facilidade de implementação e transferência de objetos na notação JSON (JavaScript Object Notation), fazendo com que ela seja facilmente integrada a outros serviços.

Enquanto Web Services usam normalmente o XML como padrão para trocar dados entre aplicações diferentes, o DataSnap REST, no Delphi, utiliza o padrão JSON para a troca de informações.

JSON é uma notação baseada em texto usada para representar objetos JavaScript, de modo que eles possam ser persistidos ou

transferidos de um aplicativo para outro. Enquanto há alguns anos atrás era consenso usar o XML como uma notação para representar estruturas complexas de dados, nos últimos anos, JSON ganhou popularidade por ser mais compacto, mais ligado a conceitos de linguagem de programação e muito rápido de ser analisado (parser) por aplicações baseadas em browser com JavaScript. Assim como o XML, ele pode ser usado pela maioria das linguagens de programação, devido a um conjunto crescente de bibliotecas e ferramentas. Veja o comparativo a seguir na **Listagem 1**.

Listagem 1. Comparativo de tamanho entre o XML e o JSON

JSON – 284 caracteres

```
{
   "titulo":"Tenda dos Milagres",
    "autor":"Jorge Amado",
    "paginas":320,
    "ano":2008,
    "editora":
 {
        "nome":"Companhia das Letras",
        "site":"www.ciadasletras.com.br"
    }
    "livrarias":[
        {
            "nome":"Saraiva",
            "site":"www.saraiva.com",
            "preco":50.00
        },
        {
            "nome":"Submarino",
            "site":"www.submarino.com.br",
            "preco":44.99
     }
    ]
}
```

XML – 404 caracteres

```
<livro>
    <titulo>Tenda dos Milagres</titulo>
    <autor>Jorge Amado</autor>
    <paginas>320</paginas>
    <ano>2008</ano>
    <editora>
        <nome>Companhia das Letras</nome>
        <site>www.ciadasletras.com.br</site>
    </editora>
    <livrarias>
        <livraria>
         <nome>Saraiva</nome>
         <site>www.saraiva.com</site>
         <preco>50.00</preco>
        </livraria>
        <livraria>
         <nome>Submarino</nome>
         <site>www.submarino.com.br</site>
         <preco>44.99</preco>
        </livraria>
    </livrarias>
</livro>
```

É bem visível a diferença na quantidade de caracteres entre os dois formatos, o XML tem a característica de ser mais verboso, os atributos são descritos de forma a facilitar a leitura humana. Já o JSON possui um foco diferenciado, o que lhe permite expressar as mesmas informações com um tamanho reduzido, ou seja, de forma menos verbosa.

DataSnap REST Server Application

O framework DataSnap do Delphi vem se firmando cada vez mais como um grande concentrador de regras de negócio. Ele possui arquitetura segura, alta performance, é escalável, atende e gerencia chamadas de diversos tipos de clientes, como aplicações no próprio Delphi, C++ Builder, HTML5 Builder, iOS, MacOS, Windows Phone, BlackBerry, Android e JavaScript. É isso mesmo, é possível criar um servidor DataSnap com regras de negócio pré-estabelecidas e disponibilizar recursos/métodos para vários tipos de clientes diferentes e, ainda, contar com recursos como a autenticação e a autorização, cache, callback, entre outros.

HTML Client DataSnap REST Application

Talvez, a seguinte dúvida possa ser levantada: se há tantas opções para criar uma aplicação web com o Delphi, como vimos anteriormente, por que devo usar esta abordagem com o DataSnap?

Vamos a algumas vantagens:

- Montagem de toda a interface usando apenas recursos de HTML, JavaScript e CSS. Assim, fica mais fácil dividir a aplicação entre aqueles responsáveis pelo layout e aqueles que cuidam da implementação dos recursos;
- Por ser um cliente em um padrão aberto, não é necessária uma licença de Delphi para montar o layout. É possível desenvolver toda a estrutura de apresentação em um editor HTML de sua preferência, como Dreamweaver, Notepad++, entre outros, o que facilita a terceirização deste serviço;
- É possível utilizar todos os recursos atuais disponíveis para o desenvolvimento web como HTML5, jQuery, CSS.

Desvantagem: requer conhecimento de HTML, JavaScript etc. Caso o desenvolvedor realmente ache que isso é uma desvantagem pode ser que o mesmo ainda precise de um tempo para se adaptar ao desenvolvimento de aplicações web.

Principais funcionalidades disponíveis no DataSnap

Sessões de usuário

Sabemos que requisições web são stateless, ou seja, uma nova requisição funciona como se fosse a primeira vez que é enviada para o servidor. Quando o usuário insere informações e passa para a próxima página, os dados serão perdidos e o usuário não será capaz de recuperar essa informação. Então, como é possível criar carrinhos de compra, por exemplo? A sessão (session) fornece um mecanismo para armazenar informações sobre a memória do servidor (**Figura 1**).

Sempre que um cliente chama um método de servidor pela primeira vez, o mecanismo do DataSnap cria uma nova sessão para este cliente. As solicitações subsequentes deste cliente passam então a incluírem um cabeçalho com o sessionID, fazendo com que o

servidor possa reconhecer se o pedido vem do mesmo cliente. Uma session no DataSnap pode suportar tipos primitivos como inteiro, string, boolean, double, ou qualquer outro tipo de objeto, inclusive, nossos próprios objetos personalizados. Para cada cliente, os dados da sessão são armazenados separadamente, o que significa que não há riscos de misturar os dados de um pedido de um cliente com os de outro. A classe TDSSessionManager é responsável pelo gerenciamento de sessões no DataSnap. Usando o método estático GetThreadSession é possível recuperar os dados da sessão do cliente atual. A sessão é válida até que expire (por padrão, 20 minutos) ou que seja fechada explicitamente (método CloseSession).

Será apresentado a seguir um exemplo simples do uso de sessões nas **Listagens 2** e **3** respectivamente.

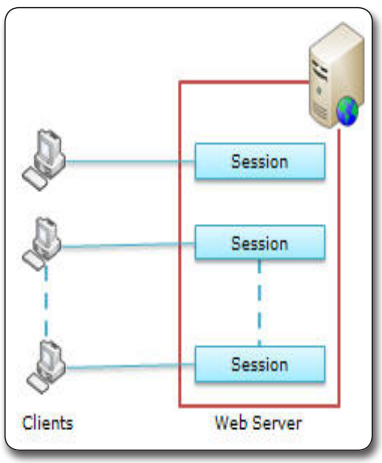

Figura 1. Exemplo de funcionamento das sessões do DataSnap

Listagem 2. Declaração de métodos no servidor DataSnap

```
01 TServerMethods1 = class(TDSServerModule)
02   public
03     { Public declarations }
04     function EchoString(Value: string): string;
05     ...
06     procedure SetValue(AValue: string);
07     function GetValue: string;
08   end;
```

Listagem 3. Implementação dos métodos criados na Listagem 2

```
01 function TServerMethods1.GetValue: string;
02 var
03   session: TDSSession;
04 begin
05   session := TDSSessionManager.GetThreadSession;
06   // recupera os dados da sessão de uma chave chamada "myKey"
07   Result := session.GetData('myKey');
08 end;

09 procedure TServerMethods1.SetValue(AValue: string);
10 var
11   session: TDSSession;
12 begin
13   session := TDSSessionManager.GetThreadSession;
14   // grava os dados da sessão em uma chave chamada "myKey"
15   session.PutData('myKey', AValue);
16 end;
```

Observe que a **Listagem 2** apenas exibe a declaração dos métodos no servidor de aplicação. A **Listagem 3** ilustra a implementação dos métodos GetValue e SetValue. Na linha 5 é utilizada a variável session que recebe o método GetThreadSession da classe TDSSessionManager. O resultado é retornado através do método GetData passando o valor myKey (nome da chave armazenada no método setValue) – linha 07. Ainda na **Listagem 3,** nota-se que é utilizado mesmo método GetThreadSession, porém, aqui é armazenado o valor da sessão usando o método PutData (linha 15).

Autenticação

É natural e muitas vezes desejável que em uma aplicação web façamos uso de algum tipo de controle de acesso de usuários. Por isso, o DataSnap oferece uma funcionalidade de autenticação e autorização para os métodos de servidor de forma simples e robusta.

A autenticação é solicitada toda vez que o servidor recebe um pedido de um cliente não autenticado. Após o usuário e sua sessão serem autenticados este procedimento não é chamado novamente até que o usuário saia do sistema (feche sua sessão) ou tenha sua sessão expirada.

Para isso, é possível codificar o evento UserAuthenticate do componente TDSAuthenticationManager, onde o parâmetro **valid**, passado como referência, sinaliza um usuário autenticado ou não, conforme a regra de autenticação a definir. Veja na **Listagem 4**.

Listagem 4. Método UserAuthenticate do componente TDSAuthenticationManager

```
01 procedure TWebModule2.DSAuthenticationManager1UserAuthenticate(Sender:
     TObject; const Protocol, Context, User, Password: string; var valid:
     Boolean; UserRoles: TStrings);
02 begin
03   valid := (User = 'embarcadero') and (Password = 'delphi');
04 end;
```

Podemos notar que a variável **valid** receberá o valor True caso os parâmetros User e Password sejam preenchidos com os valores 'embarcadero' e 'delphi' respectivamente, e o valor False para qualquer outro tipo de informação (linha 03).

O exemplo é simples e está sendo usado apenas para fins didáticos. De forma geral, para validar os usuários e suas senhas, você deve consultar um banco de dados ou um arquivo de configuração ou até mesmo um mecanismo de autenticação do Sistema Operacional.

O exemplo do DataSnap REST baseado em browser esconde a interface de usuário padrão (aquela janela com user e password do próprio navegador) e exibe um formulário de Login baseado em HTML.

Quando informar o usuário e a senha, será possível ter acesso às funcionalidades restantes da página HTML. Porém, se a página for atualizada, as informações da sessão serão perdidas e será preciso realizar o login novamente (o que convenhamos, não é muito interessante). Para resolver isso, podemos facilmente substituir o código JavaScript inicial, no método onLoad()

de cada página HTML. Sendo assim, o login será necessário somente se ainda não há uma sessão válida. Veja o resultado conforme a **Listagem 5**.

Listagem 5. Evento onLoad em JavaScript para tratar a sessão

```
01 function onLoad()
02 {
03  showTime();
04  initSessionData(true);
05  loginRequired = (getSessionID() == null);
06  setConnection('<#host>','<#port>','<#urlpath>');
07  if (loginRequired)
08  {
09   showLogin(true);
10  }
11  else
12  {
13   showLogin(false);
14  }
15 }
```

Na linha 05 é criada uma variável loginRequired que recebe o valor True caso não haja sessão existente. Na linha 07 é realizado o teste desta variável, exibindo o login ou não de acordo com o valor obtido (linhas 08 a 14).

Também foi criado um método JavaScript nesta página para fechar a sessão do usuário quando ele clicar em um HTML Button, por exemplo, como frequentemente vemos em Internet Banks e Web Mails (**Listagem 6**).

Listagem 6. Método Sair em JavaScript

```
01 function Sair()
02 {
03  serverMethods().executor.closeSession();
04  AdminInst = null;
05  onLoad();
06 }
```

Papéis e autorização

Inicialmente, qualquer usuário autenticado pode executar qualquer método exposto pelo servidor. Entretanto, não é raro nos depararmos com situações onde é preciso conceder privilégios diferentes a cada usuário ou a grupos/funções de usuários específicos. Uma simples alternativa na arquitetura DataSnap é o suporte a papéis de usuário.

É possível, na autenticação do usuário, atribuir um ou mais papéis a este usuário, fazendo uso do parâmetro UserRoles do evento OnUserAuthenticate do componente TDSAuthenticationManager, conforme ilustra a **Listagem 7.**

O código realiza o mesmo procedimento de validação de usuário mencionado anteriormente, porém, aqui é verificado se os mesmos foram preenchidos (linha 03). Caso tenham sido preenchidos, a variável **valid** receberá o valor True, sendo testada logo a seguir (linha 04). Observe aqui que o parâmetro UserRoles é do tipo TStrings, sendo assim, a lista de usuários pode ser atribuída a este parâmetro. A linha 06 atribuí o valor 'stand' à UserRoles. Por último, na linha 07, é verificado se o usuário (parâmetro de entrada user) equivale a 'embarcadero', adicionando à UserRoles então o valor 'admin' (linha 09), que seria o privilégio deste usuário.

Para determinar quais papéis podem fazer uso de um método do servidor de aplicação, basta decorarmos cada um dos nossos métodos (ou toda a classe) com o atributo TRoleAuth. Observe a **Listagem 8.**

Listagem 7. Adicionando papéis ao usuário

```
01 procedure TWebModule2.DSAuthenticationManager1UserAuthenticate(Sender:
   TObject; const Protocol, Context, User, Password: string; var valid:
   Boolean; UserRoles: TStrings);
02 begin
03   valid := (User <> '') and (Password <> '');
04   if valid then
05   begin
06     UserRoles.Add('stand');
07     if user = 'embarcadero' then
08     begin
09       UserRoles.Add('admin');
10     end;
11   end;
12 end;
```

Listagem 8. Decorando os métodos do servidor de aplicação

```
01 uses Datasnap.DSAuth;

02 type
03   TServerMethods1 = class(TDSServerModule)
04   public
05     { Public declarations }
06     [TRoleAuth ('stand')]
07     function EchoString(Value: string): string;
08     [TRoleAuth ('admin')]
09     function ReverseString(Value: string): string;
10   end;
```

Nesse exemplo foi feito com que o método EchoString (linha 07) esteja disponível para qualquer usuário autenticado em nossa aplicação (linha 06), enquanto o método ReverseString (linha 09) só está disponível por usuário membros da função **admin (**linha 08).

É possível, ainda, filtrar os métodos disponíveis, dependendo do usuário, codificando o evento OnUserAuthorize do componente AuthenticationManager. Esta é uma poderosa forma de definir critérios específicos dentro do seu modelo de autorização.

Filtragem

Uma funcionalidade interessante disponível no modelo DataSnap REST é a capacidade de filtrar o resultado (strings longas ou um conjunto de dados complexo). Há dois filtros disponíveis: substring (ss) e tabela (t). Ambos possuem três operações disponíveis: count (c), offset (o), e range(r). Estes elementos podem ser adicionados como parâmetros de URL. Confira o uso dos dois tipos de filtros com base nos métodos de servidor descritos na **Listagem 9.**

No exemplo, será chamado o método EchoString e estabelecida uma filtragem para que ele retorne apenas os dois primeiros caracteres do seu resultado, conforme a URL exibida na linha 01 da **Listagem 10**. O resultado será o equivalente a linha 02 da mesma listagem.

Listagem 9. Definição de métodos do servidor de aplicação

```
01 type
02  TServerMethods1 = class(TDSServerModule)
03  ...
04  public
05   { Public declarations }
06   function EchoString(Value: string): string;
07   function GetDados: TDBXReader;
08  end;
```

Listagem 10. URL de chamada no método EchoString realizando filtragem

```
01 http://localhost:8080/datasnap/rest/TServerMethods1/EchoString/delphi/?ss.c=2

02 {"result":["de"]}
```

Há também um método GetDados (também da **Listagem 9**), que retorna um TDBXReader (**Nota do DevMan 1**) como resultado. Que seria um SELECT a uma tabela de banco de dados, sendo este, um resultado de um conjunto de dados mais complexo. Neste caso é retornada a faixa contando em cinco (5) registros a partir do terceiro registro (item 2), por isso, t.r=2,5 (linha 01), como exibido na **Listagem 11**. O resultado neste caso seria o demonstrado na linha 02.

Nota do DevMan 1

TDBXReader é uma classe do Framework dbExpress que fornece um leitor unidirecional para um conjunto de linhas de base de dados. TDBXReader é retornado por TDBXCommand.ExecuteQuery. O método TDBXReader.Next é utilizado para acessar a primeira linha e caso haja, as linhas sucessivas no ResultSet. Os valores recuperados podem ser acessados usando a propriedade Value que é um array. A propriedade Value é sobrecarregada e podemos utilizar tanto a posição da coluna quanto seu nome.

Ter um montante parcial de dados para cada pedido tornaria mais fácil para paginar um grande conjunto de resultados do cliente HTML, retornando, por exemplo, dez registros por vez e criar assim, várias páginas de dados. Um exemplo de como implementar estas chamadas diretamente através do código HTML/JavaScript é ilustrado na **Listagem 12**.

Entre as linhas 01 e 06 da **Listagem 12**, temos a representação do método ReverseString em JavaScript. Note que na linha 03 a variável **myVal** recebe o valor do campo chamado **valueField**, contido na página. A linha 04 faz a chamada do método no servidor de aplicação DataSnap através de REST, passando como parâmetro o valor da variável **myVal** e atribuindo o resultado à variável '**s**'. A seguir, na linha 04, o resultado armazenado em '**s**' é atribuído novamente o elemento **valueField**, através de **getElementById**. Já a partir da linha 09 a linha 30, é realizado o tratamento do método **onGetDados**. Observe que na linha 11 é montado o valor correspondente que será passado ao servidor de aplicação. Na linha 12 a variável **myTable** recebe o conteúdo do elemento chamado de **tbResult**. Posteriormente, na linha 13, é feita a requisição ao método **GetDados** do servidor de aplicação DataSnap, montando a URI e atribuindo os parâmetros de acordo com as variáveis anteriores. Entre as linhas 14 e 29, simplesmente é realizado um laço contado (for) no conteúdo obtido através do resultado do servidor de aplicação. Este conteúdo é lido e interpretado através de colunas (<td>) e linhas (<tr>), montando assim uma tabela para a exibição dos dados. A linha 29 apenas realiza o incremento do número da página.

Listagem 11. URL da chamada no método GetDados

```
01  http://localhost:8080/datasnap/rest/TServerMethods1/GetDados/?t.r=2,5

02  {"result":[{"table":[["ContactID",6,0,0,0,4,4,0,false,false,0,false,false],
    ["EmailAddress",26,1,0,0,102,100,0,true,false,0,false,false],["FirstName",
    26,2,0,0,102,100,0,false,false,0,false,false],["LastName",26,3,0,0,102,100,0,
    false,false,0,false,false]],"ContactID":[3,4,5,6,7],"EmailAddress":
    ["kim2@adventure-works.com","humberto0@adventure-works.com",
     "pilar1@adventure-works.com","frances0@adventure-works.com",
     "margaret0@adventure-works.com"],"FirstName":["Kim","Humberto",
     "Pilar","Frances","Margaret"],"LastName":["Abercrombie","Acevedo",
     "Ackerman","Adams","Smith"]}]}
```

Listagem 12. Exemplo de implementação das chamadas DataSnap REST com JavaScript

```
01 function onReverseStringClick()
02 {
03   var myVal = document.getElementById('valueField').value;
04   var s = serverMethods().EchoString(myVal, null, {"ss.c": "2"});
05   document.getElementById('valueField').value = s.result;
06 }

07 var pageNumber = 0;
08 var pageSize = 10;

09 function onGetDados()
10 {
11    var myRange = (pageNumber * pageSize) + "," + pageSize;
12    var myTable = document.getElementById('tbResult');
13    var s = serverMethods().GetDados(null, {"t.r": myRange});

14    var strHeadTable = '<tr>';
15    for (var n=0; n < s.result.table.length; n++) {
16      strHeadTable = strHeadTable + '<th>' + s.result.table[n][0] +
       '</th>';
17    }
18    strHeadTable = strHeadTable + '</tr>'
19    for (var rowTable=0; rowTable < s.result.ContactID.length;
      rowTable++) {
20       var strRowTable = '<tr>';
21       strRowTable += '<td>' + s.result.ContactID[rowTable] + '</td>';
22       strRowTable += '<td>' + s.result.EmailAddress[rowTable] +
        '</td>';
23       strRowTable += '<td>' + s.result.FirstName[rowTable] + '</td>';
24       strRowTable += '<td>' + s.result.LastName[rowTable] + '</td>';
25       strRowTable += '</tr>';
26       strHeadTable += strRowTable;
27    }
28    myTable.innerHTML = strHeadTable;
29    pageNumber++;
30 }
```

Conclusão

O DataSnap nos oferece implementações do que há de mais moderno para Web, temos gerência de sessão, controle de autenticação, autorização, suporte à REST e muito mais. Como podemos ver, temos uma poderosa ferramenta em nossas mãos. Ainda há alguns recursos como Callback e Cache que podem ser abordados em outras oportunidades, e que certamente darão aos seus projetos web ainda mais flexibilidade e desempenho. Agora é por a mão na massa!

Marcelo Varela

marcelo@logicatreinamentos.com.br

Embarcadero MVP (Most Valuable Professional), Delphi Developer Certification 7/2005/2006/2007/XE

Master Delphi Developer Certification, Delphi Instructor Certification. Palestrante na Delphi Conference Brasil 2009, 2010, 2011, 2012, BorCon 2005, 2006 e 2007. 12 anos em experiência em desenvolvimento de software em Delphi, professor de Sistemas de Informação do IFRN, pós-graduado em Desenvolvimento de Sistemas Web pela UnP e Bacharel em Sistemas de Informação pela UnP

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

CURSOS ONLINE

A Revista Clube Delphi oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

CONHEÇA ALGUNS DOS CURSOS:

- Delphi para Iniciantes
- Criando componente Boleto em Delphi
- Loja Virtual em Delphi Prism, Rave Reports – Impressão de Gráficos Personlizados
- Aplicação para repasse de Dados com Delphi

Para mais informações :

www.devmedia.com.br/cursos/delphi

(21) 3382-5038

Relatórios Dinâmicos com FastReport

Utilizando o editor do FastReport em tempo de execução

Quando desenvolvemos uma aplicação comercial, um fator que muitas vezes é determinante como um diferencial em relação a outras aplicações do mercado, pode ser a diversidade de relatórios disponíveis na aplicação. O cliente não deseja apenas o que é "padrão" no mercado, mas sim poder deixar aquela aplicação de acordo com o seu gosto, na medida do possível é claro. Normalmente o principal ponto onde podem ser observadas estas características, está nas mais diversas impressões que podem existir em um software, partindo desde o logotipo da empresa até a cor da fonte de determinado campo em um relatório.

Não só por ordem de customização, mas muitas vezes, existe a necessidade de fazer a impressão de um determinado item de várias formas possíveis e que não podem ser previstas com antecedência.

A modificação de um relatório pode ser um trabalho complicado e cansativo e a gerência dessas modificações merece atenção, principalmente em geradores que criam um arquivo de relatório para cada versão diferente do mesmo. Nesse artigo abordaremos esse assunto, como minimizar esse impacto criando um exemplo simples que permite a customização de um layout de relatórios, sem exigir a criação de vários arquivos modelo.

A versão 4 do FastReport traz consigo um recurso muito interessante que possibilita abrir um editor dentro de um simples TPanel, por exemplo. Isso é feito através do componente TfrxDesigner, que não é acessível na versão distribuída junto com o Delphi XE 2. Essa versão disponibilizada pela Embarcadero não possui todos os recursos da versão completa do FastReport.

O componente TfrxDesigner permite a utilização de vários recursos para a criação e manutenção de arquivos .fr3 que é a extensão dos arquivos de template dos relatórios do FastReport. Com esse componente é possível criar um editor completo, exatamente igual ao editor de relatórios que usamos sempre quando criamos um relatório pelo FastReport, já que o próprio editor do FastReport é um formulário feito em Delphi e que não depende de nenhuma biblioteca externa para funcionar.

Resumo DevMan

Relatórios Dinâmicos com FastReport :

O FastReport é a nova ferramenta padrão para a criação de relatórios no Delphi XE, é muito utilizada e possui uma infinidade de recursos muitas vezes desconhecidos pela maioria dos desenvolvedores. Nesse artigo abordaremos um recurso que é a utilização do FastReport não só na sua forma convencional, mas também podendo utilizar seu próprio editor em tempo de execução para ajustes simples, como no caso de posições de campos em um relatório. No artigo será feito um exemplo onde, através desse recurso, o usuário poderá criar templates de impressão de cheques com posições variadas de campos para cada folha de cheque e de cada banco específico, utilizando como padrão o mesmo relatório do FastReport. Esse exemplo simples abrirá as portas para a sua criatividade, tornando infinitamente mais simples fazer determinadas tarefas que necessitam de dinamismo ou customização de relatórios.

Vale lembrar que esse recurso pode permitir até a criação de relatórios inteiros em tempo de execução e o armazenamento dos layouts dos mesmos em qualquer meio alternativo como, por exemplo, arquivos XML ou melhor, em um banco de dados, facilitando e reduzindo muitos cuidados que devem ser considerados ao manter arquivos externos nos aplicativos.

Em que situação o tema é útil:

O tema é útil para demonstrar uma saída viável para quando existe a necessidade de criar diversos templates para uma mesma impressão, onde o usuário poderá selecionar o template desejado para a impressão. Dessa forma, com apenas um relatório padrão será possível atender a várias peculiaridades principalmente no que se diz respeito à posição dos campos na impressão.

Outro recurso interessante que será apresentado no artigo, é algo que permite armazenar as definições do relatório no próprio dfm do seu componente container, seja este um TForm ou um TDataModule, evitando assim a necessidade de criação de um arquivo .fr3 para o relatório que será criado. É um recurso muito interessante, quando não queremos manter arquivos externos para os relatórios do sistema.

Com a versão completa do FastReport 4 instalada no Delphi XE 2, criaremos um exemplo onde será possível a criação de modelos de cheques para impressão em diferentes tipos de folha de diversos bancos diferentes. Esse recurso poderá ser utilizado em uma situação real, onde é feita impressão de cheques a partir de alguma conta a pagar, por exemplo. Isso resolve um problema recorrente, onde cada cliente que utiliza o produto, faz uso de um banco diferente com folhas de cheques específicas. Essa situação é considerada um problema, pois, muitas vezes obrigaria o desenvolvedor a optar por tratar cada caso em particular, consequentemente gerando um retrabalho sempre que surgisse outro banco ainda não implementado. Vale lembrar, que além dos recursos vistos aqui, o FastReport ainda conta com uma gama de outros recursos interessantes como o FastScript e o FastReport Exports (**Nota do DevMan 1**).

Nota do DevMan 1

O FastReport possui uma vasta gama de recursos inexplorados pela maioria dos desenvolvedores, como por exemplo o FastScript, que permite a criação de scripts em várias linguagens de programação diferentes, tornando os softwares mais flexíveis. Outro recurso também interessante é o FastReport Exports, que permite a exportação de relatórios do FastReport para formatos como XLS, RTF, HTML, CSV entre outros. Na seção Links desse artigo é possível encontrar uma referência a sua documentação, explore esse manual e observe sua variedade de recursos.

Construindo um exemplo

Nosso exemplo se constituirá a partir de um cadastro simples de modelos de cheques que serão salvos em um banco de dados Firebird. Estes serão acessados sempre que for necessário carregar o template salvo e realizar a impressão do cheque.

Para criar um novo projeto VCL acessamos o menu File>New>VCL Forms Application – Delphi. Depois de criado, renomeamos o Form1 para frmPrincipal e salvamos o projeto em uma pasta de sua escolha. A Unit1.pas foi denominada de ufrmPrincipal e o Project1.dpr foi chamado de prjDesigner.dpr.

Para armazenar os modelos de cheque utilizaremos uma estrutura simples de banco de dados com apenas uma tabela. No exemplo foi adotada a versão 2.1 do Firebird e o IBExpert para a criação e manuseio do banco de dados.

No IBExpert criaremos um novo banco de dados acessando o menu Database>Create Database, conforme mostrado na **Figura 1.**

Uma tela com as informações necessárias para a criação do banco de dados será exibida. Preencha seus respectivos valores conforme descrito na **Tabela 1.**

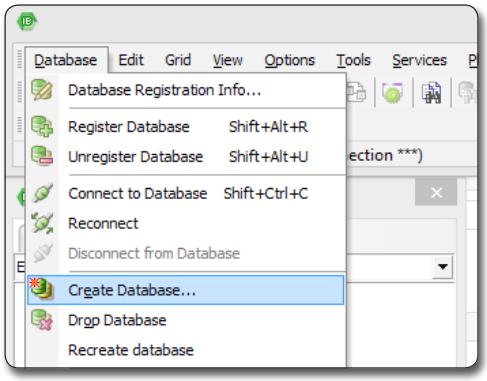

Figura 1. Criando um banco de dados no IBExpert

Campo	Valor
Server name	Localhost
Database	C:\BANCO.FDB
Client Library File	fbclient.dll
Username	SYSDBA(ou o seu usuário do Firebird)
Password	masterkey(ou sua senha do Firebird)

Tabela 1. Parâmetros para a criação do banco de dados

Após preenchê-los, clique em OK. Uma tela para registrar o banco de dados no IBExpert será exibida, informe qual é a sua versão do Firebird e forneça um Alias (apelido) para o seu banco de dados, assim, será mais simples encontrá-lo no IBExpert. Por último, clique em Register.

Dê um duplo clique em cima do banco de dados recém-criado (ele encontras-se na palheta Databases, geralmente localizada do lado esquerdo da tela. Caso não esteja sendo exibida, basta acessar o menu View>DB Explorer) para conectar-se, em seguida, acesse o menu Tools>Script Executive e execute o script SQL da **Listagem 1.**

Listagem 1. Script SQL do Banco de dados

```
CREATE TABLE MODELO_CHEQUE (
ID              INTEGER NOT NULL,
DESCRICAO  VARCHAR(30) NOT NULL,
MODELO      BLOB SUB_TYPE 0 SEGMENT SIZE 16384
);
ALTER TABLE MODELO_CHEQUE ADD CONSTRAINT PK_MODELO_CHEQUE
PRIMARY KEY (ID);
CREATE GENERATOR GEN_MODELO_CHEQUE;
SET GENERATOR GEN_MODELO_CHEQUE TO 0;
```

Note que no script da **Listagem 1** que foi criada uma tabela chamada MODELO_CHEQUE e um generator chamado GEN_MODELO_CHEQUE, que será utilizado mais adiante para gerar os Ids dos modelos dos cheques.

Nossa tabela possui três campos, ID - que será a chave primária da tabela, DESCRICAO - que será utilizado para atribuir um nome aos modelos criados e MODELO, do tipo BLOB, que irá armazenar as configurações dos modelos em um formato XML.

Com nosso banco de dados criado, voltaremos ao projeto no Delphi XE2 para criarmos uma conexão com o mesmo. Para esse exemplo será utilizada a suíte dbExpress que é muito adotada entre os desenvolvedores Delphi e que permite a conexão com os mais variados tipos de SGDBs disponíveis no mercado. Sendo assim, no frmPrincipal adicione um componente TSQLConnection e configure-o com as propriedades definidas na **Tabela 2.**

Propriedade	Valor
Name	SQLConn
ConnectionName	ModeloCheque
Driver	Firebird
LoginPrompt	False
VendorLib	fbclient.dll
LibraryName	dbxfb.dll
Params['Database']	localhost:C:\BANCO.FDB
Params['User_Name']	Sysdba
Params['Password]	Masterkey

Tabela 2. Configurações da conexão com o banco de dados

Com tudo configurado, altere a propriedade Connected para True para testar se a conexão está configurada corretamente. Depois de testar altere a propriedade para False já que utilizaremos ClientDataSets e não é necessário deixar a conexão com o banco aberta, delegando essa responsabilidade ao DataSetProvider que irá retornar dados apenas quando necessário.

Agora criaremos um CRUD (operações de inserção, alteração, exclusão e localização de dados) simples para manusearmos os modelos de cheques criados. Adicione ao frmPrincipal um componente TSQLQuery e altere suas configurações conforme a **Tabela 3.**

Propriedade	Valor
Name	qryModeloCheque
SQL	SELECT ID, DESCRICAO, MODELO FROM MODELO_CHEQUE ORDER BY DESCRICAO
SQLConnection	SQLConn

Tabela 3. Configurações do TSQLConnection

O componente qryModeloCheque contém o comando SQL que será executado no banco para a recuperação dos modelos cadastrados. Depois de configurado, dê um duplo clique sobre ele e no Fields Editor, clique com o botão direito, selecionando a opção Add all fields, como mostrado na **Figura 2.**

Note que os três campos da tabela foram adicionado à lista. Após este processo, iremos configurá-los para que o DataSetProvider consiga fazer o CRUD sem maiores problemas. Selecione então o campo ID e na propriedade ProviderFlags configure a opção pfInKey para True, altere também a propriedade Required para False, pois, esse campo não será exibido na tela, e, somente quando for ser persistido no banco de dados receberá um ID.

Agora adicione ao frmPrincipal um componente TDataSetProvider e configure-o conforme a **Tabela 4.**

Propriedade	Valor
Name	dspModeloCheque
DataSet	qryModeloCheque
Options[poPropogateChanges]	True
UpdateMode	upWhereKeyOnly

Tabela 4. Configurações do TDataSetProvider

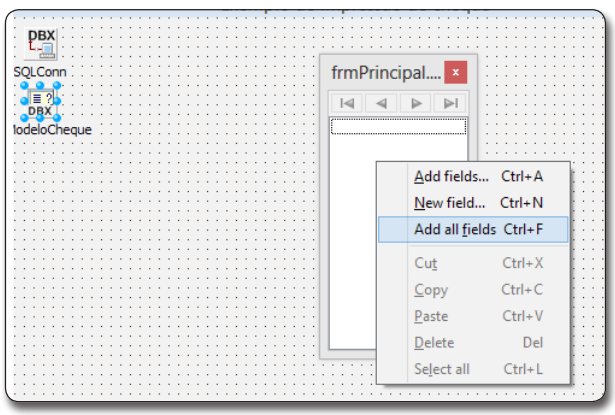

Figura 2. Adicionando os campos a qryModeloCheque

A propriedade poPropogateChanges deve ser alterada para True pois, como geraremos o ID do registro no DataSetProvider e esperamos que o ClientDataSet receba o novo ID, qualquer alteração no registro que ocorra no DataSetProvider será automaticamente repassada ao ClientDataSet. Já UpdateMode é responsável por informar ao DataSetProvider como ele deverá montar as cláusulas de UPDATE e DELETE. Nesse caso foi selecionada a opção upWhereKeyOnly onde o dspModeloCheque irá montar as sentenças SQL baseadas nos campos com a propriedade pfInKey com valor True, ou seja, o campo ID.

Agora adicione ao frmPrincipal um componente TClientDataSet e configure-o de acordo com a **Tabela 5.**

Propriedade	Valor
Name	cdsModeloCheque
ProviderName	dspModeloCheque

Tabela 5. Configurações do TClientDataSet

Após configurá-lo, abra o fields editor do cdsModeloCheque, clique com o botão direito do mouse e selecione Add all fields. Os campos do componente qryModeloCheque devem aparecer na lista. Configure o campo ID da mesma forma que o campo ID do componente qryModeloCheque foi configurado e feche o fields editor.

Feito isso, criaremos uma rotina para gerar o id automaticamente, sempre que um novo registro for adicionado. Nossa rotina irá executar uma instrução SQL no banco de dados e retornará o id do registro que está sendo criado. Adicione para isso o código da **Listagem 2**.

A função GetIdByGenerator recebe como parâmetro o nome do Generator (linha 01). A seguir, é criada uma query dinâmica através da variável qry do tipo TSQLQuery (linhas 03 a 06). Na linha 07 a conexão é atribuída à variável, que executa em seguida um Select na tabela RDB$Database (tabela do Firebird que possui apenas um registro com parametrizações do SGBD), conforme visto na linha 08. Esta instrução chama o método GEN_ID do banco de dados, que tem por objetivo incrementar o Generator (**Nota do DevMan 2**). O valor é automaticamente retornado através do Result da função, como mostra a linha 10. Por último, na linha 12, a variável é destruída e liberada da memória.

Listagem 2. Listagem da função que gera o id do registro a partir do generator

```
01 function TfrmPrincipal.GetIdByGenerator(GeneratorName: String): Integer;
02 var
03   qry: TSQLQuery;
04 begin
05   try
06     qry := TSQLQuery.Create(SQLConn);
07     qry.SQLConnection := SQLConn;
08     qry.SQL.Text := Format('SELECT (GEN_ID(%s, 1)) AS ID FROM
       RDB$DATABASE',[GeneratorName]);
09     qry.Open;
10     Result := qry.FieldByName('ID').AsInteger;
11   finally
12     FreeAndNil(qry);
13   end;
14 end;
```

Nota do DevMan 2

Generators e outros elementos de um banco de dados Firebird são armazenados nas chamadas tabelas de sistemas, que podem ser identificadas pelo sufixo rdb$. A **Figura A** ilustra as tabelas de sistema existentes e como elas se relacionam.

Com o relacionamento montado, fica fácil extrair informações sobre a estrutura de um banco de dados. Para exemplificar, quando criamos uma nova tabela em um banco de dados, elas são armazenadas na tabela rdb$relations e por padrão, são ligadas a outros objetos como colunas, views, índices, triggers, constraints e verificações de segurança (check constraints). A **Figura B** mostra como as tabelas envolvidas estão relacionadas.

Sendo assim, para recuperar os campos de uma tabela, é possível executar a sentença SQL a seguir:

```
SELECT rdb$field_name, rel_fld.rdb$field_name,
  rdb$field_type, rdb$field_sub_type,
  rel_fld.rdb$null_flag, rdb$field_length, rdb$field_scale,
  rdb$character_length, rdb$field_precision,
  fld.rdb$default_source, fld.rdb$validation_source
FROM rdb$relations rel
  JOIN rdb$relation_fields rel_fld
    ON rel_fld.rdb$relation_name = rel.rdb$relation_name
  JOIN rdb$fields fld
    ON rel_fld.rdb$field_source = fld.rdb$field_name
WHERE rel.rdb$relation_name = : NomeDaTabela
ORDER BY rel_fld.rdb$field_position, rel_fld.rdb$field_name
```

A tabela principal é a rdb$relations, que recebeu o apelido de "rel". Foram criados os joins (ligações) necessários para que se pudesse extrair todas as informações. Na cláusula WHERE é utilizado o nome da tabela, armazenado no campo relation_name. Uma vez localizado o registro que contém esse nome, achou-se a tabela e através dos joins formados, a informação sobre sua estrutura é trazida.

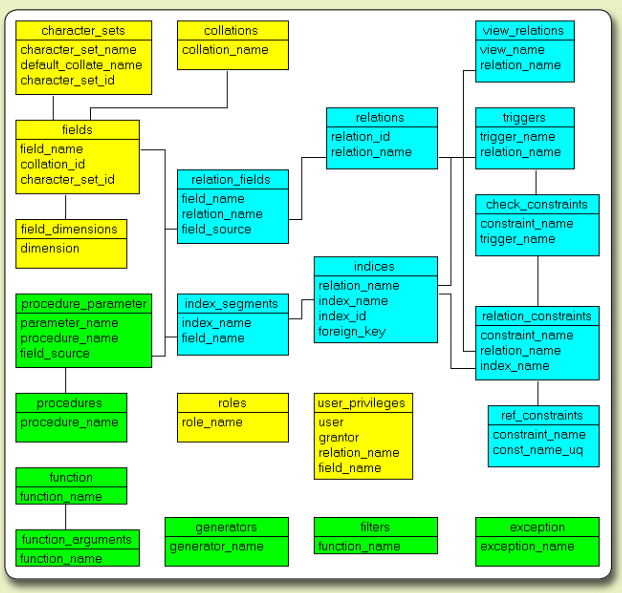

Figura A. Tabelas de sistema de um banco Firebird (versões 1 – 2)

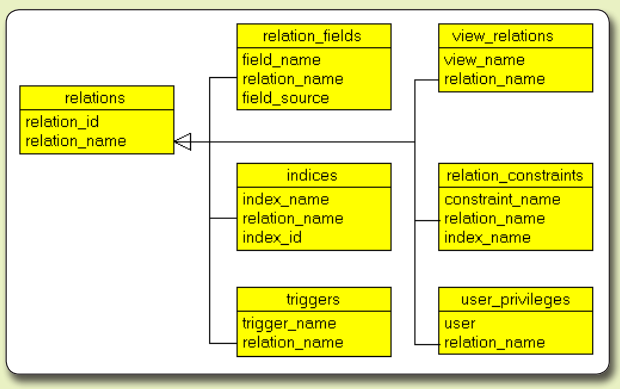

Figura B. Tabelas envolvidas na criação de uma tabela.

Depois de criar a função GetIdByGenerator, iremos implementar o código do evento BeforeUpdateRecord do dspModeloCheque, conforme a **Listagem 3.**

Esse evento é executado antes de salvar cada registro que foi alterado no ClientDataSet sempre que o método ApplyUpdates é chamado. Observe que primeiramente é testado se a origem das informações vem da qryModeloCheque (linha 03), a seguir, se o tipo de alteração é ukInsert (inserção – linha 05). Caso seja, só então é chamado o método GetIdByGenerator, atribuindo o novo ID para o registro que está sendo inserido (linha 07).

Depois de criarmos o mecanismo necessário para que nosso CRUD gere automaticamente o ID dos novos registros, vamos programar nosso cdsModeloCheque para sempre manter as informações sincronizadas com o banco de dados. Insira o código da **Listagem 4**.

Listagem 3. Listagem da implementação do método BeforeUpdateRecord do dspModeloCheque

```
01 procedure TfrmPrincipal.dspModeloChequeBeforeUpdateRecord(Sender:
    TObject; SourceDS: TDataSet; DeltaDS: TCustomClientDataSet; UpdateKind:
    TUpdateKind; var Applied: Boolean);
02 begin
03  if SourceDS = qryModeloCheque then
04  begin
05   if UpdateKind = ukInsert then
06   begin
07    DeltaDS.FieldByName('ID').NewValue :=
      GetIdByGenerator('GEN_MODELO_CHEQUE');
08   end;
09  end;
10 end;
```

Listagem 4. Métodos para aplicar as alterações no Banco de Dados

```
01 procedure TfrmPrincipal.cdsModeloChequeAfterDelete(DataSet: TDataSet);
02 begin
03  cdsModeloCheque.ApplyUpdates(0);
04 end;

05 procedure TfrmPrincipal.cdsModeloChequeAfterPost(DataSet: TDataSet);
06 begin
07  cdsModeloCheque.ApplyUpdates(0);
08 end;

09 procedure TfrmPrincipal.cdsModeloChequeAfterCancel(DataSet: TDataSet);
10 begin
11  cdsModeloCheque.CancelUpdates;
12 end;
```

Os eventos AfterDelete e AfterPost do cdsModeloCheque (entre as linhas 01 e 08), executam o método ApplyUpdates para que quando um registro seja alterado ou excluído, o dspModeloCheque aplique essas alterações no banco de dados. No evento AfterCancel da mesma do cdsModeloCheque, o método CancelUpdates é chamado limpando qualquer alteração que esteja no cache do ClientDataSet e que foi cancelada pelo usuário.

Agora criaremos a interface do nosso cadastro. Para isso, adicione um TDataSource, um TDBNavigator, um TDBEdit e uma TLabel e configure-os para que trabalhem de acordo com a necessidade do cadastro. Como isso é uma configuração simples, não serão abordadas aqui, porém, no código-fonte disponibilizado no portal será possível ver detalhadamente. Depois de configurados, o frmPrincipal deverá ficar semelhante a **Figura 3.**

Altere a propriedade Active do cdsModeloCheque para True para testar se o ClientDataSet está configurado corretamente, assim, finalizamos o código necessário para manter um CRUD simples.

Criando o mecanismo de cheque

Como este artigo trata-se de um exemplo, vamos supor que nosso cheque terá apenas três campos, sendo eles o valor do cheque, o valor por extenso e a data do cheque. Deste modo, primeiramente adicione outro TClientDataSet e renomeie ele para cdsCheque. Abra o fields editor do cdsCheque e crie os campos conforme as configurações da **Tabela 6.**

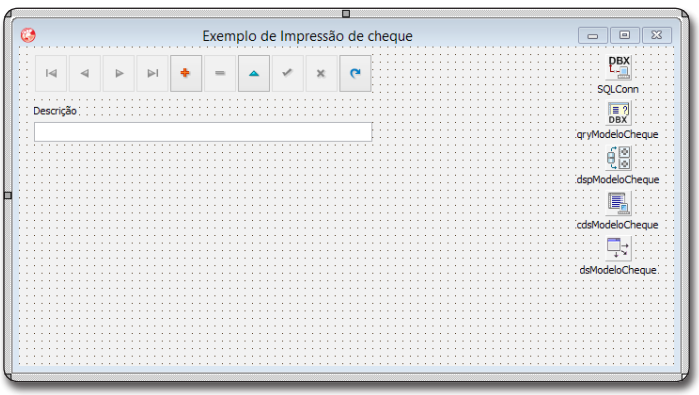

Figura 3. Tela Principal

Name	Type	Size
ID	Integer	(Padrão)
VALOR	String	20
EXTENSO	String	100
DATA	String	100

Tabela 6. Campos do cdsCheque

Após criados os campos, selecione o ID e altere sua propriedade AutoGenerateValue para arAutoInc, assim, os IDs serão adicionados automaticamente, e por fim, clique com o botão direito sobre o cdsCheque e selecione CreateDataSet. Essa será a estrutura de cada cheque que será impresso. No campo VALOR será armazenado o valor do cheque formatado como Real (R$), no campo Extenso será escrito o valor do cheque por extenso e no campo data, a data do cheque por extenso. Agora vamos criar o relatório do FastReport com o modelo padrão para a impressão do nosso cheque.

Para utilizarmos um DataSet em um relatório do FastReport, é necessária uma interface para essa comunicação. O componente TfrxDBDataSet é responsável em fazer essa comunicação entre os DataSets do Delphi e os componentes do FastReport. Na verdade, ele trabalha como se fosse um "adaptador", permitindo que um relatório do FastReport utilize um DataSet do Delphi.

Adicione um componente TfrxDBDataSet ao frmPrincipal e configure-o conforme a **Tabela 7.**

A propriedade DataSet faz referência a classe TDataSet (que é base de todos os componentes de dados), sendo assim, é possível adicionar tanto um ClientDataSet, ADO, IBX como qualquer outro. A propriedade Name representa o nome do componente no Delphi, enquanto a propriedade UserName representa o nome desse componente dentro do relatório do FastReport (nome de exibição).

Propriedade	Valor
DataSet	cdsCheque
Name	frxCdsCheque
UserName	frxCdsCheque

Tabela 7. Configurações do TfrxDBDataSet

Para criar um relatório no FastReport, temos o componente TfrxReport, onde é possível abrir o editor do FastReport e criar relatórios.

Adicione ao frmPrincipal um componente TfrxReport e altere sua propriedade Name para frxCheque, em seguida, dê um duplo clique sobre ele para criarmos o nosso relatório. É importante ressaltar que não manteremos nenhum arquivo fr3, pois, como já foi dito anteriormente, as definições do nosso relatório serão armazenadas no . dfm do frmPrincipal, o componente frxCheque possui um propriedade chamada StoreInDFM que é responsável por fazer esse controle. Com essa opção ativada, todas as definições do relatório são mantidas no arquivo dfm do formulário e compiladas juntas com a aplicação, assim não é necessário carregar um arquivo fr3 para imprimir um relatório que já esteja carregado no dfm.

Com o editor do FastReport aberto, primeiro iremos marcar o frxCdsCheque, para que possamos utilizar seus campos no nosso relatório. Através do menu Relatório>Dados, selecione o frxCdsCheque e clique em OK. Veja que agora ele está disponível para ser usado no relatório. Desenhe o relatório de forma que fique semelhante ao apresentado na **Figura 4.**

Observe que os componentes foram renomeados para poderem ser identificados com clareza. Atente-se aos nomes dos objetos exibidos na árvore de dados do relatório e renomeie os componentes do seu relatório da mesma forma, conforme a **Tabela 8.**

Agora criaremos nosso Editor para permitir a alteração das posições dos campos na tela. Adicione ao formulário um componente TfrxDesigner e nomeie-o para frxDesigner. Acrescente um TPanel, altere sua posição Align para alBotton e deixe a propriedade Caption em branco. O componente frxDesigner permite trabalhar com o editor do FastReport em tempo de execução.

O primeiro método a ser visto é o ArmazenarModelo, conforme ilustrado na **Listagem 5**.

O objetivo do código é realizar o armazenamento do report no componente. Nota-se que primeiramente é verificado se a variável FModeloReport está nula, encerrando a execução do método caso a afirmação seja verdadeira (linhas 03 e 04). Caso contrário, a variável é utilizada para que seja criado um MemoryStream e inicializado na posição 0 (linhas 05 e 06), armazenando-o no componente frxCheque, através do método SaveToStream (linha 07). A seguir, na **Listagem 6** é apresentado o método AtualizaModelo.

Nome anterior	Novo nome
Page1	Pagina
MasterData1	Cheque
frxCdsChequeVALOR	CampoVALOR
frxCdsChequeEXTENSO	CampoEXTENSO
frxCdsChequeDATA	CampoDATA

Tabela 8. Nomes dos componentes do relatório

Listagem 5. Método ArmazenarModelo

```
01 procedure TfrmPrincipal.ArmazenarModelo;
02 begin
03  if Assigned(FModeloReport) then
04   Exit;
05  FModeloReport := TMemoryStream.Create;
06  FModeloReport.Position := 0;
07  frxCheque.SaveToStream(FModeloReport);
08 end;
```

Listagem 6. Método AtualizaModelo

```
01 procedure TfrmPrincipal.AtualizarModelo;
02 var
03  Designer: TfrxDesignerForm;
04 begin
05  if Assigned(frxCheque.Designer) then
06  begin
07   frxCheque.Designer.Free;
08   frxCheque.Designer := nil;
09  end;
10  FModeloReport.Position := 0;
11  if cdsModeloChequeMODELO.IsNull then
12   frxCheque.LoadFromStream(FModeloReport)
13  else
14   CarregarReport;
15  frxCheque.DesignReportInPanel(Panel1);
16  Designer := (frxCheque.Designer as TfrxDesignerForm);
17  Designer.DockBottom.Visible := False;
18  Designer.DockTop.Visible := False;
19  Designer.ObjectsTB1.Visible := False;
20  Designer.LeftDockSite1.Visible := False;
21  Designer.LeftDockSite2.Visible := False;
22  Designer.RightDockSite.Visible := False;
23 end;
```

O código é responsável por atualizar o modelo sempre que o usuário navegar entre os já existentes, ou quando um novo for criado. Entre as linhas 05 a 09 há uma verificação para garantir que qualquer designer chamado anteriormente seja destruído. Em seguida, entre as linhas 10 e 14 é determinado qual modelo será carregado, se é um modelo do banco de dados ou o modelo padrão. Na linha 15 é possivel encontrar a chamada ao método Design-ReportInPanel. Esse é o responsável em criar o editor do FastReport e exibi-lo dentro do Panel que é passado como parâmetro. Na linha 16 temos um TypeCast do tipo TfrxCustomDesigner para um objeto do tipo TfrxDesignerForm. Isso é feito pois o método

DesignReportInPanel cria um objeto desse tipo. A instancia desse objeto é um editor que foi feito completamente em Delphi e pode ser facilmente visualizado segurando a tecla Ctrl e clicando em cima da classe TfrxForm-Designer e pressionando F12. Entre as linhas 17 a 23 está o código que oculta todos os painéis do editor do FastReport.

A **Listagem 7** ilustra o método ApagarModelo que como o próprio nome sugere, realiza a exclusão do modelo.

Listagem 7. Método ApagarModelo

```
01 procedure TfrmPrincipal.ApagarModelo;
02 begin
03  if Assigned(frxCheque.Designer) then
04  begin
05   frxCheque.Designer.Free;
06   frxCheque.Designer := nil;
07  end;
08  FreeAndNil(FModeloReport);
09 end;
```

Na linha 01 é checado se há conteúdo carregado no Designer do componente frxCheque. Sendo assim, apenas é realizada a limpeza do Designer (linhas 05 e 06) e consequentemente, da variável (linha 08).

Os métodos SalvarReport e CarregarReport são encarregados de salvar as definições do Relatório no campo MODELO do cdsModeloCheque e depois recuperá-lo da mesma forma para alterações futuras. Ambos são descritos respectivamente nas **Listagens 8** e **9**.

Listagem 8. Método SalvarReport

```
01 procedure TfrmPrincipal.SalvarReport;
02 var
03  TempStream: TMemoryStream;
04 begin
05  try
06   TempStream:= TMemoryStream.Create;
07   TempStream.Position := 0;
08   frxCheque.SaveToStream(TempStream);
09   TempStream.Position := 0;
10   cdsModeloChequeMODELO.
      LoadFromStream(TempStream);
11  finally
12   FreeAndNil(TempStream);
13  end;
14 end;
```

O método SalvarReport da **Listagem 8**, simplesmente cria um MemoryStream (linha 06), iniciando-o em 0 (linha 07 e salva as definições do relatório (frxCheque) dentro da variável (linha 08), para a seguir, o ser passado para o campo MODELO do cdsModeloCheque (linha 10). Na linha 12 esta variável é finalizada e liberada da memória.

Já o método CarregarReport (**Listagem 9**) faz o processo inverso, primeiramente verifica se o modelo não está nulo (linha 05 - quando é o caso de novos registros), a seguir, é criado e inicializada uma variável Memory Stream temporária para manuseio das configurações do relatório (linhas 08 e 09). Posteriormente, na linha 10, as informações do cdsModeloCheque são carregadas no Stream temporário para enfim, na linha 12, serem repassadas para o relatório através do método LoadFromFile. A linha 12 destrói e libera a variável da memória.

Listagem 9. Método CarregarReport

```
01 procedure TfrmPrincipal.CarregarReport;
02 var
03  TempStream: TMemoryStream;
04 begin
05 if cdsModeloChequeMODELO.IsNull then
06   Exit;
07  try
08   TempStream:= TMemoryStream.Create;
09   TempStream.Position := 0;
10   cdsModeloChequeMODELO.
      SaveToStream(TempStream);
11   TempStream.Position := 0;
12   frxCheque.LoadFromStream(TempStream);
13  finally
14   FreeAndNil(TempStream);
15  end;
16 end;
```

Finalmente, implementaremos a chamada dos métodos anteriormente definidos. Desta vez, serão utilizados para manipulação apenas os eventos do cdsModelo e o OnDestroy do formulário. Por serem apenas as chamadas, neste caso não há necessidade de aprofundarmos, sendo assim, a **Listagem 10** ilustra os respectivos eventos.

Testando o Editor

A partir de agora, o editor deve estar funcionando sem maiores problemas. Para finalizar o exemplo iremos simular a impressão de alguns cheques. Porém, antes serão necessários alguns ajustes finais. Primeiramente definiremos um método chamado de AddChequeImpressao, conforme mostra a **Listagem 11.**

O método simplesmente atribui os valores passados como parâmetro ao componente cdsCheque (linhas 04 a 06), fazendo reuso deste para a impressão dos cheques.

Para finalmente testar o projeto, adicione um TButton ao frmPrincipal,

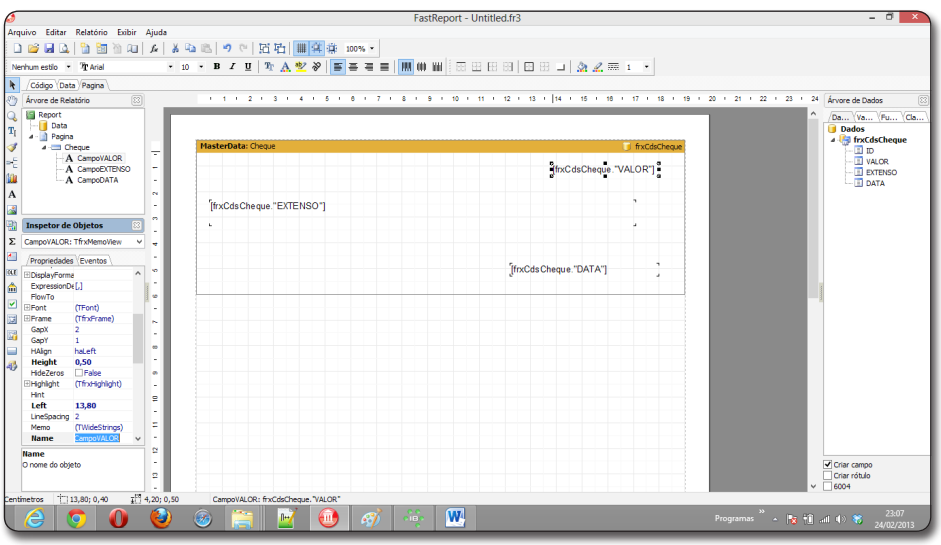

Figura 4. Resultado final do modelo de cheque para o frxCheque

altere sua propriedade Caption para Preview e Name para btnPreview, implementando no evento OnClick do botão o código da **Listagem 12.**

Listagem 10. Eventos correspondentes as chamadas dos métodos anteriores

```
[Evento OnAfterScroll do cdsModeloCheque]
01 procedure TfrmPrincipal.cdsModeloChequeAfterScroll(DataSet: TDataSet);
02 begin
03  ArmazenarModelo;
04  AtualizarModelo;
05 end;

[Evento OnBeforePost do cdsModeloCheque]
01 procedure TfrmPrincipal.cdsModeloChequeBeforePost(DataSet: TDataSet);
02 begin
03  SalvarReport;
04 end;

[Evento OnAfterCancel do cdsModeloCheque]
01 procedure TfrmPrincipal.cdsModeloChequeAfterCancel(DataSet: TDataSet);
02 begin
03  cdsModeloCheque.CancelUpdates;
04  AtualizarModelo;
05 end;

[Evento OnDestroy do formulário]
01 procedure TfrmPrincipal.FormDestroy(Sender: TObject);
02 begin
03  ApagarModelo;
04 end;
```

Listagem 11. Método AddChequeImpressao

```
01 procedure TfrmPrincipal.AddChequeImpressao(Valor: Real; Data: TDate);
02 begin
03  cdsCheque.Append;
04  cdsChequeVALOR.AsString := FloatToStrF(Valor, ffCurrency, 10, 2);
05  cdsChequeEXTENSO.AsString := valorPorExtenso(Valor);
06  cdsChequeDATA.AsString := FormatDateTime(' dd "de" MMMM "de" yyyy',
    Data);
07  cdsCheque.Post;
08 end;
```

Listagem 12. Evento OnClick do botão de preview

```
01 procedure TfrmPrincipal.btnPreviewClick(Sender: TObject);
02 var
03  i: Integer;
04 begin
05  cdsCheque.CreateDataSet;
06  for i := 1 to 10 do
07   AddChequeImpressao(1000 / i, Date);
08  frxCheque.ShowReport;
09  cdsCheque.Close;
10 end;
```

Observe que 10 cheques são adicionados para a impressão através de uma estrutura de repetição "for" (linha 06) que chama o método AddChequeImpressao passando valores como parâmetro (linha 07). Na linha 08 o método ShowReport é executado, exibindo o relatório, para finalmente, na linha 09, o cdsCheque ser fechado, perdendo seus dados em memória.

A partir de agora já é possível testar nosso exemplo executando o projeto. Altere a posição dos campos na tela, salve o registro e clique no botão Preview para visualizar a impressão do relatório com os campos bem onde foram configurados, conforme sua necessidade. Observe também, que para cada modelo criado, é possível "arranjar" os campos na tela de forma diferente, possibilitando criar um modelo de impressão para folhas de cheque de qualquer banco de acordo com a necessidade.

Dessa forma seria possível fazer uma rotina de impressão de cheques onde o usuário poderia escolher em um DBComboBox o modelo de cheque que deseja imprimir e, a partir disso, bastaria ler as definições do modelo e atualizar o relatório antes de imprimir.

Conclusão

O artigo buscou abordar de uma forma simplificada um recurso do FastReport que não é muito explorado pelos desenvolvedores e que pode propor uma solução simples para problemas recorrentes do dia a dia. Sem este recurso, seria necessário realizar a criação de um arquivo .fr3 para cada modelo de cheque, pois, sempre que um novo modelo fosse requisitado, haveria de se criar novos arquivos fr3. Com o recurso de armazenar as definições do relatório nos arquivos .dfm dos projetos, talvez não seja mais necessário manter os arquivos externos fr3, de forma a deixar todo conteúdo necessário a aplicação contido no próprio arquivo exe ou em uma possível modularização do sistema, através de bpl's por exemplo.

O componente frxDesigner possui ainda vários outros recursos que valem a pena serem estudados, como por exemplo, um recurso de restrições, onde é possível limitar determinadas ações no editor aos usuários, como não inserir novos objetos ao relatório, não criar novas paginas no relatório entre outros. Vale a pena utilizar a documentação do FastReport como guia de referência e aumentar os conhecimentos sobre esse componente, pois, mais cedo ou mais tarde, ele poderá te poupar um grande trabalho.

Wanderley Donizeti Panosso Junior
wanderley_panosso@hotmail.com
Cursando Análise de Desenvolvimento de Sistemas pela FATEC-TQ. Atua como analista e programador de sistemas da equipe de desenvolvimento Delphi da Siplan Control-M.(www.siplancontrolm.com.br).

Documentação FastReport
http://www.fast-report.com/pt/product/fast-report-vcl/documentation/

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Padrões de Arquitetura de Aplicações Corporativas

Desenvolvendo a camada de lógica com qualidade

Uma das maiores causas de enxaqueca nos desenvolvedores está fortemente ligada a uma má arquitetura adotada no início do desenvolvimento de uma aplicação, um caminho adotado, que inicialmente parecia ser o mais fácil se tornou um grande pesadelo. Todo projeto de software deve começar por uma boa arquitetura, pois é essa etapa que guiará todo o ciclo de vida do projeto, se ela for mal projetada, consequentemente o projeto será mau feito. Existe uma série de fatores que deve sempre ser levado em conta ao se iniciar um projeto, uma delas é a verificação e análise da arquitetura adotada, mas infelizmente em muito dos casos não se chegar nem a se projetar antes de desenvolver. Simplesmente tem-se algumas conversas básicas e vão logo com a mão na massa. Esse fator é bastante comum em empresas pequenas, onde se busca um retorno o quanto antes, contudo, sem pensar no futuro. Um pouco de tempo gasto inicialmente no projeto com planejamentos arquitetônicos de software, economizará tanto dinheiro como o tempo gasto para manter o mesmo projeto.

Até quando os desenvolvedores e empresas não vão perceber que a melhor solução dos seus problemas está antes de iniciar o projeto? Será que ainda terão que perder muito dinheiro e tempo para que percebam isso?

Infelizmente a realidade dos desenvolvedores é essa, na verdade podemos dizer que é do ser humano, que só muda de atitudes quando sente no bolso o peso de suas más decisões. Um bom projeto sempre começa com um ótimo planejamento.

O grande pesadelo

Desenvolver software não é difícil, a questão está no fato de que um software pode possuir um ciclo de vida longo, o que existe um desenvolvimento de qualidade, que se adapte às mudanças por esse tempo. É por isso que surgem padrões de projeto e metodologias de processo de software, como a CMMI (**Nota do DevMan 1**), que buscam respectivamente tornar um software mais adaptável e garantir a qualidade do de desenvolvimento.

Resumo DevMan

Padrões de Arquitetura de Aplicações Corporativas:

Este artigo traz técnicas de arquiteturas de aplicações corporativas, que são úteis para a organização da lógica de negócio. Os princípios e boas práticas mostrados neste artigo trarão grandes resultados no desenvolvimento de aplicações corporativas ricas e complexas e sem contar que auxiliarão o leitor a aplicar essas técnicas na prática, com exemplos do dia a dia utilizando conceitos de orientação a objeto.

Em que situação o tema é útil:

Este tema é útil para os desenvolvedores que querem aprimorar o desenvolvimento de sistemas de grande porte utilizando técnicas para organizar a lógica de forma adequada, preparando-a para futuras alterações e também para os desenvolvedores que ainda não tem conhecimento sobre o assunto e querem aprender. Este artigo lhe auxiliará desde o planejamento da regra de negócio quanto na execução de técnicas para organizar as mesmas no projeto. Este artigo é voltado para o desenvolvimento de aplicações ricas e complexas, mas não limitando a elas, podendo ser também utilizada em projetos pequenos e simples.

Nota do DevMan 1

CMMI é o acrônimo de Capability Maturity Model Integration, que possui as melhores práticas necessárias para o desenvolvimento e maturidade do software. Integra práticas definidas pelo modelo CMM, criado em 1980 como um modelo de avaliação de risco para empresa de software pelo Departamento de Defesa dos Estados Unidos.

Você já perguntou o motivo de toda vez que um cliente pede uma alteração em uma funcionalidade os programadores sentem uma tremenda dor de cabeça no momento em que começam a alterá-la? Muitas das vezes o segredo de um sono tranquilo está ligado à forma em que programamos o nosso software, somos a causa dos nossos pesadelos, nós somos culpados dos nossos erros. Vamos entender melhor a solução do nosso problema, ou melhor, a solução para diminuir a nossa dor de cabeça.

Arquitetura

O termo arquitetura abordado desse artigo, nada mais é do que um conjunto de técnicas ou experiências vividas por um grupo de desenvolvedores, que são compartilhadas. Segundo Ralph Johnson "A arquitetura se resume em coisas importantes que se devem fazer e para isso precisam ser tomadas decisões difíceis, caso essas decisões passam a ser mais fáceis do que se propôs, não é arquitetura". Ou seja, o termo arquitetura empregado nada mais é do que uma tomada de decisão.

O análise arquitetural nada mais é do que o análise dos problemas enfrentados e que demandam uma série de tomadas de decisões, para que possamos projetar uma arquitetura que possa prolongar ou manter o ciclo de vida do projeto em questão.

Nós desenvolvedores não devemos ser apenas programadores, mas antes devemos ser grandes arquitetos de soluções. O grande problema hoje é que só queremos nos preocupar com a lógica do negócio em si e como programar, mas esquecemos de que antes mesmos de construir um prédio tem-se a necessidade de se firmar os alicerces.

Neste artigo vamos falar desses alicerces, basicamente de três padrões de arquitetura: Roteiro de Transição, Modelo de Tabela e Modelo de Domínio. Também são tratados outros, necessários para o entendimento, mas o foco não são eles.

Roteiro de transação

A grosso modo, um roteiro de transação (**Figura 1**) nada mais é do que a separação de solicitações em métodos, ou seja, um método chamado FinalizarCompra é o responsável por efetuar todo o processo de finalização de compras. Esse método é o responsável por separar o produto, por diminuir a quantidade do produto no estoque, por enviar os dados para enviar o produto para o cliente e muitas das vezes ele é até o responsável por persistir todas essas informações no banco de dados.

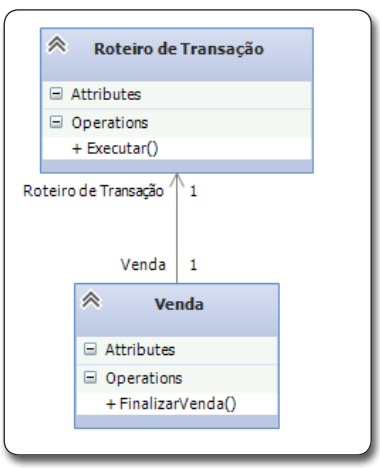

Figura 1. Roteiro de Transação

O Roteiro é um dos padrões mais encontrados na arquitetura de muitos projetos de software, por sua simplicidade e o baixo requerimento de conhecimento por parte do desenvolvedor. O Roteiro de Transação é recomendado quando o desenvolvedor tem uma lógica de negócio simples e requer certa rapidez para o projeto. Esse padrão pode ser considerado como uma mão dupla, pois tanto ele pode beneficiar, quanto ele pode lhe dar muita dor de cabeça, pois a lógica do Roteiro de Transação é separada por roteiro (ou funcionalidade) e não pelo contexto da regra de negócio, e isso, poderá levar uma duplicação do código.

Se no projeto existirem dois roteiros que necessitam saber o valor da compra de um determinado cliente, em cada roteiro tende-se a ter um código que efetuará as devidas buscas no banco e os devidos cálculos, e isso é desaconselhado, pois trará redundância de código. Porém, existem situações onde o Roteiro de Transação se encaixa muito bem, veremos mais a frente como implementá-lo em um projeto e os devidos cuidados que se deve ter com o mesmo.

Modelo de domínio

Modelo de Domínio (**Figura 2**) é um modelo de objetos que efetua e agrupa dados ou conjunto de objetos, no caso esses objetos devem possuir o mesmo contexto para que possam seguir um dos princípios chaves da orientação a objetos, que é a abstração. Essa é um processo mental para lidar com a complexidade, onde se omite detalhes não essenciais, assim, se obtém a simplificação de algo complexo.

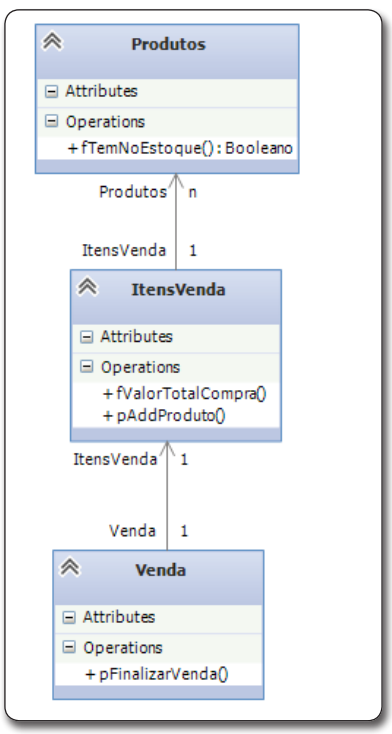

Figura 2. Modelo de Domínio

Podemos considerar o Modelo de Domínio como um gerador de camadas lógicas, que agrupam vários objetos separados por contexto, ou seja, existem objetos que trabalham simplesmente

como repositórios, ou que representam o próprio dado do banco, ou ainda encapsulam as regras de negócio.

Seu uso é o mais aconselhado, isto devido a uma separação melhor de responsabilidade, facilitando assim a manutenção futura do projeto. Mas por outro lado ele traz uma complexidade a mais, pois com a organização contextual, uma operação de negócio pode se estender por vários objetos e contemplar toda a comunicação entre os objetos envolvidos pode ser complexo. Diante desses conceitos vem uma grande pergunta: Quando devo utilizar o Modelo de Domínio? Esse modelo é aconselhado para projetos mais robustos, que concentraram uma grande massa de dados e esses dados são frequentemente manipulados e possuem regras de negócios complexas. Resumindo, ao pensar na realidade de nossos sistemas, podemos dizer que em praticamente qualquer projeto comercial o Modelo de Domínio pode ser utilizado, exceto para os projetos que são bastante simples e exigem certa rapidez no desenvolvimento.

Modelo de Tabela

Modelo de Tabela (**Figura 3**) nada mais é do que um modelo de objetos que representa uma determinada tabela, onde o mesmo é o responsável por efetuar todo o processo de manipulação da mesma e das linhas que a compõem.

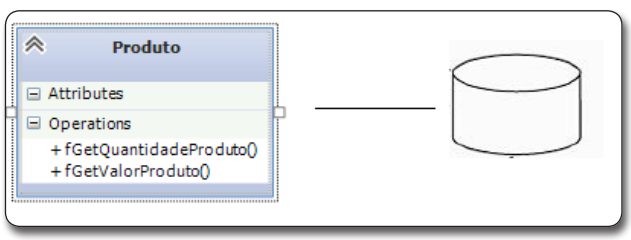

Figura 3. Modelo de Tabela

Funciona como empacotamento de registro e ações que podem ser exercidas sobre uma massa de dados. Esse modelo tenta amenizar o contraste entre o banco relacional e os objetos. Na prática os objetos que se adequam a esse modelo funcionam como uma representação de uma tabela, podendo serem feitos os mesmos processos que se podem fazer direto na tabela.

Frequentemente se usa o Modelo de Tabela quando se quer ter, ou se necessita estrutura interna de objetos como uma estrutura tabular, cujos objetos (registros) geralmente são obtidos mediante a uma consulta SQL. Devido ao nome do modelo ser Modelo de tabela, existe uma certa confusão quanto à forma de sua aplicação, achando-se que deve se criar um modelo para cada tabela, e só, mas é bastante útil deixar bem claro que um Modelo de Tabela não é só para representar uma tabela, mas para representar uma coleção de registros do banco de dados, seja tabela, view (**Nota do DevMan 2**) ou até resultado de consultas de banco de dados.

Gateway de Dados

É necessário uma abordagem sobre o Gateway de Dados para que se possa entender melhor os padrões aqui apresentados. Um Gateway nada mais é do que um encapsulador de acessos externos que o sistema realiza. Por acesso externo podemos compreender a conexão com um banco de dados, mas poderá também ser um arquivo XML ou qualquer outra fonte de dados. No nosso caso será uma tabela do banco de dados, sendo responsável por fazer todo o processo do banco de dados na tabela referente aos produtos (**Figura 4**).

Sua atuação vai de consultas simples, como saber o valor de um determinado produto até a persistência de dados no banco.

Nota do DevMan 2

A view nada mais é do que um objeto do banco de dados, que não armazenam dados, seus registros são compostos dinamicamente por uma consulta (SQL). Diferentemente das tabelas, as views não são objetos físicos, não ocupam espaço em discos, pois seus dados são simplesmente resultados de consultas previamente otimizadas.

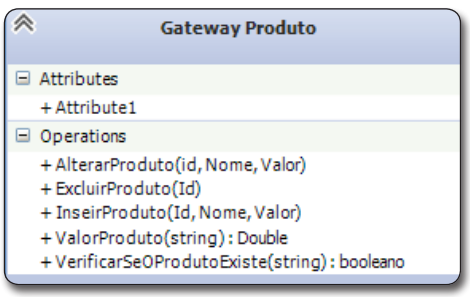

Figura 4. Diagrama de classe de um Gateway de Produtos

Conjunto de Registros

Um Conjunto de Registros nada mais é do que uma representação em memória dos dados tabulares. Em muitos casos é um resultado de uma consulta SQL do banco de dados retornado em forma de um objeto.

Um Conjunto de Registros é exatamente o resultado de uma consulta. O Delphi já faz todos esse processo, basta utilizarmos alguns dos mecanismos que ele oferece, que pode ser o DBExpress, ou IBExpress, o ADO e fora os componentes de terceiros que também podemos utilizar além da possibilidade de criarmos algo próprio. Um conjunto de registro é indicado quando se utiliza o Modelo Tabela para organizar a lógica de Domínio, pois o mesmo precisa de um ambiente que trabalhe bem com dados tabulares, que é o caso do Modelo Tabela.

Os três padrões e um dilema

Já é de se notar que o Roteiro de Transação é o menos indicado dos três pois dá a possibilidade ou a margem para que o desenvolvedor cometa algumas p´raáticas ruins, ou os chamados "bad smell" (**Nota do DevMan 3**) no desenvolvimento, que no caso é a duplicação de código.

O grande dilema fica entre o Modelo de Tabela e o Modelo de Domínio, ambos tem as suas vantagens e também as suas desvantagens. O Modelo de Tabela ele é muito bom para tra-

balhar com estrutura tabulares, que na maioria dos casos é assim que encontramos, mas a grande desvantagem do Modelo de Tabela é justamente o fato de ele representar tão bem uma tabela, que perde grandes vantagens e poder que os objetos nos proporcionam. Esse modelo é aconselhado, quando se tem a necessidade de aproveitar ao máximo a estrutura tabular que um banco de dados proporciona, e onde a regra de negócio não seja tão complexa.

Nota do DevMan 3

O termo bad smell em inglês é mau cheiro. Na programação se utiliza bastante esse termo para dizer que um código é ruim, que precisa de uma refatoração. Geralmente os códigos que possuem duplicidade e que possuem alta complexidade, alto acoplamento, são chamados de bad smell.

Já o Modelo de Domínio oferece todo o poder que um objeto pode proporcionar ao desenvolvedor, como polimorfismo, relacionamento entre objetos e sem contar que trabalha muito bem com regras de negócio muito complexas, sem perder as vantagens de uma manutenção mais ágil e fácil, mas devido a sua complexidade muitos dos desenvolvedores não o implementam e muito dos que tentam implementar, por falta de conhecimento, implementam de forma errada, assim, o que seria a solução dos seus problemas passa a ser causa das suas enxaquecas.

Problema abordado

Foi criado um servidor DataSnap, que conterá toda a regra de negócio de uma determinada corporação, este servidor possuirá as regras de negócio de uma loja qualquer. A função do servidor é de saber o valor total dos produtos que foram comprados pelo cliente e também de finalizar a venda e saber se a venda foi finalizada com sucesso ou não.

O foco desse artigo é de ajudar a projetar uma arquitetura para o pequeno problema apresentado e não de programar o código que efetuará todo esse processo, em nenhum momento desse artigo será mostrado como debitar do estoque um produto, ou como salvar essas informações no banco de dados.

Separação em camadas

É muito importante frisar que este artigo está mostrando técnicas para organizar a camada Model, ou seja, estes padrões apresentados nesse artigo têm por finalidade de organizar a camada Model (**Nota do DevMan 3**) para que a mesmo esteja receptível para possíveis mudanças futuras. Uma regra de negócio bem organizada é sinal de menos dor de cabeça futura.

Roteiro de Transação

Já conhecendo o problema que será abordado vamos primeiramente resolvê-lo usando o Roteiro de Transação. Na **Listagem 1** está uma classe chamada TVenda que é a responsável por fazer todo o processo de uma venda, desde uma simples verificação, igual a do nosso problema, até a finalização de uma compra, ou seja, todo o processo relacionado diretamente com o contexto de TVenda, ou seja, da venda, deverá ficar nessa classe.

Nota do DevMan 3

Em um sistema com boa arquitetura encontramos algumas divisões lógicas que visam facilitar a manutenção e adição de novos recursos. Essas são as camadas.

Para tornar a explicação sobre as camadas mais intuitiva imagine um sistema desktop. Com toda certeza nele encontramos um formulário de cadastro. O seu formulário representa a camada de apresentação do sistema, isto porque é nela que os dados são apresentados. Se o sistema é cliente/servidor e procedural, a sua camada de domínio ou model, de base dados e até mesmo serviços podem ser representadas pelos Data Modules existentes e seus componentes de acesso ao banco de dados. Por último temos uma camada global que pode ser representada pelo seu conjunto de funções e procedimentos públicos.

Continuando com o paralelo a um sistema comum, desenvolvido por um programador Delphi, nos formulários nós fazemos acesso direto aos Data Modules, aos ClientDataSets, geramos SQL, tudo sem preocupação alguma. Quando essas camadas se misturam é dito que existe um acoplamento entre elas. No momento que você insere código SQL em evento OnClick de um botão o acoplamento foi realizado. Seu formulário que é sua camada de apresentação, está fazendo acesso direto aos recursos do que seria sua camada de dados. Isso impede a reutilização da lógica contida no formulário. Tudo isso dificulta a manutenção do seu sistema porque você terá SQL espalhado por todo código e não apenas centralizado em uma área. Ou seja, nós misturamos o que seriam nossas camadas sendo que o correto é separar as camadas

Listagem 1. Classe TVenda

```
01  TVenda = class
02   public
03    function fValorTotalCompra : Double;
04    function fFinalizarVenda : Boolean;
05   end;
```

Todo o processo referente ao valor total da compra de um determinado cliente deverá ficar dentro da função fValorTotalCompra, desde acesso ao banco de dados até a lógica para fazer o somatório dos valores do produto.

Igualmente todo o processo referente ao processo de finalização da venda, desde o acesso ao banco de dados, caso for necessário, até a lógica envolvida no processo.

É recomendado criar um Gateway de Dados para a classe TVenda, esse gateway fica responsável por todo o acesso ao banco de dados e manipulação do mesmo. Um Roteiro de Transação praticamente só encapsula funcionalidades que expressam um mesmo contexto, que no caso é de uma venda. É bem simples e é indicado, como já foi dito, apenas para projetos que possuem uma regra de negócio simples, pois pode gerar uma redundância de código.

A função fValorTotalCompra faz acesso ao banco de dados na tabela de Produtos e faz a soma dos mesmos, desde que foi comprado pelo cliente, e da mesma forma a função fFinalizarVenda deverá fazer praticamente o mesmo processo para que possa salvar essas informações em uma tabela de Vendas e isso poderá induzir ao desenvolvedor a escrever códigos repetidos, mas,

nesse caso, nada que uma refatoração (**Nota do DevMan 4**) não resolva. Agora Imaginem que tenha outra classe que necessita fazer o mesmo processo ou algo parecido, mas que expressão um contexto um pouco diferente de venda, isso ainda mais induzirá ao desenvolvedor a fazer uma redundância, e nesse caso precisaria de uma refatoração um pouco mais complexa do que se for à mesma classe, podendo usar até outras técnicas como os padrões de projetos.

Nota do DevMan 4

Um dos grandes problemas encontrados em sistemas de software é evitar a deterioração tão comum dos programas durante o ciclo de vida do sistema. Essa deterioração é, geralmente, causada por modificações com objetivos de curto prazo ou por alterações realizadas sem a clara compreensão da concepção do sistema.

A refatoração (do inglês Refactoring) é o processo de modificação de software para melhorar o entendimento do código sem alterar suas funcionalidades, removendo duplicidades, melhorando a comunicação da equipe, simplificando e flexibilizando rotinas, facilitando a manutenção e evitando a inclusão de novos defeitos, aumentando assim a qualidade do produto.

O livro mais importante sobre o assunto é Refatoração - Aperfeiçoando o Projeto de Código Existente de Martin Fowler, que apresenta um catálogo com 72 refatorações, explicando seus conceitos e motivações, além de descrever detalhadamente cada uma das refatorações apresentadas.

O autor argumenta que, com treinamento apropriado, um profissional de computação pode transformar um projeto de baixa qualidade num código robusto e bem projetado, sem modificar seu comportamento observável. Outro bom argumento para a utilização dessa técnica é que qualquer um pode escrever código que um computador consiga entender, enquanto bons programadores escrevem código que pessoas conseguem entender.

A refatoração modifica um programa em pequenos passos. Assim, se um erro é cometido, torna-se mais fácil consertá-lo. Algumas oportunidades são apropriadas para aplicar refatoração. Deve-se refatorar quando for necessário construir código parecido com outro já existente, quando acrescentar novas funções, quando necessita consertar uma falha ou enquanto se revisa um código. Além disso, quando precisar escrever um comentário para melhor explicar um código, deve-se tentar refatorar primeiro.

Para finalizar o Roteiro de Transação é necessário ressaltar que caso este for o padrão de arquitetura escolhido é necessário grandes cuidados para não cometer grandes bad smells, pois esse padrão comumente induz o desenvolver cometer tais atos. Outro fator bastante importante é que este padrão poderá trazer complicações à medida que a regra de negócio vai se tornando mais complexa, pois esse padrão é indicado para regras simples.

Modelo Tabela

O Modelo Tabela já é um pouco mais complexo do que o Roteiro de Transação, mas já traz uma separação melhor da funcionalidade e isso diminuirá o risco de código redundante, mas não impedindo que isso aconteça.

Primeiramente podemos dizer que o Modelo Tabela possui uma classe para cada conjunto de registro, ou seja, para cada tabela existe uma classe que a representa e para cada view que venha a existir no banco de dados, também existirá uma tabela que a represente. Para resolver o mesmo problema agora aplicando o Modelo de Tabela, primeiramente temos que criar uma classe que representa o Produto, depois temos que criar outra classe que representa a venda em si. Podemos notar que esse processo foi oculto na exemplificação do Roteiro de Transação devido ao fato de tudo ser facilmente acessível e misturado nos métodos, isso por um lado é bom, pois é bem mais fácil de desenvolver, mas traz uma complicação para manter o código e ao longo do tempo essa complicação pode crescer até de forma exponencial.

No Modelo Tabela existe uma separação ainda melhor das funcionalidades, uma classe (**Listagem 2**) que representa a tabela Produto, que conterá as informações pertinentes ao mesmo, que ficará responsável por efetuar todo o processo de manipulação dos dados referente ao produto. Outra classe que representa a venda em questão, que conterá o código necessário para solucionarmos de fato o problema apresentado.

Listagem 2. Classe TProduto

```
TProduto = class
  public
    function fGetValorProduto(pIdProduto : string) : Double;
    function fGetQuantidadeProduto(pIdProduto : string) : Integer;
  end;
```

Podemos notar que a classe TProduto possui dois métodos que executam ações referentes a tabela produto, aonde se pede um Id para a identificação do mesmo, ou seja, encontramos conceitos de um banco de dados relacional no objeto do Modelo de Tabela. É verificado o valor do produto usando um identificador, que na verdade é a chave primária do produto no banco de dados.

A função fGetValorProduto é responsável por fazer todo o processo para pegar o valor do produto passado por parâmetro e retorná-lo para que possa ser manipulado. Já a função fGetQuantidadeProduto, que também recebe por parâmetro a identificação do produto e retorna a sua quantidade.

Vamos utilizar a mesma estrutura da **Listagem 1** para facilitar a resolução do nosso problema, mas diferente da abordagem feita no Roteiro de Transação, a classe TVenda não mais terá acesso a dados ou fará qualquer trabalho para identificar o valor do produto, para implementar o método fValorTotalCompra será feito um laço para pegar os valores dos produtos utilizando o Modelo de Tabela TProduto, efetuando um somatório. Já na implementação do método fFinalizarvenda vamos fazer também um laço, mas dessa vez será para verificar se tem o produto no estoque e caso tenha retornará true, caso contrário retornará false.

Podemos colocar métodos mais complexos no Modelo de Tabela, que retornem um conjunto de registros. Os métodos contidos no Modelo de Tabela do produto nada mais são do que uma substituição das sentenças SQL, uma mistura bem amigável entre orientação a objetos e banco relacional. Essa mistura pode continuar bem amigável pelo resto do ciclo de vida do sistema, mas desde que a regra de negócio não seja tão complexa que um Modelo de Tabela não resolva, pois o Modelo de Tabela se assemelha muito com o banco de dados relacional, mas que fez perder um pouco o poder que a orientação a objetos proporciona.

O uso de um framework ORM (**Nota do DevMan 5**) pode acelerar a aplicação dessa arquitetura, pois oferece meios automatizados da geração de SQL.

Nota do DevMan 5

ORM é o acrônimo de Object Relational Mapping, que no Brasil também é chamado de MOR (Mapeamento de Objeto Relacional). É uma técnica de mapeamento de objetos para o banco de dados que visa ser responsável de fazer todo o trabalho de persistência ao banco e dependendo do ORM, poderá até realizar consultas. Dependendo do projeto e do nível, o ORM poderá ser uma camada a mais no projeto ou poderá ser uma parte ou uma subdivisão da camada Model do projeto.

Modelo de Domínio

Dentre o Roteiro de Transação e o Modelo de Tabela, o Modelo de Domínio é o mais indicado, pois o mesmo não se assemelha tanto com o banco de dados relacional e por isso não perde o grande poder que a orientação a objetos nos proporciona, podendo ser facilmente utilizar a herança, polimorfismo, relacionamento entre objetos, etc.

Por outro lado o Modelo de Domínio traz uma complexidade ainda maior para o projeto, que requer um grande conhecimento e habilidade por parte dos desenvolvedores envolvidos. O Modelo de Domínio não é fácil de implementar, requer uma boa estratégia por parte da equipe e principalmente do arquiteto de software envolvido. Uma das grandes complexidades que o Modelo de Domínio traz é simplesmente pelo fato de misturar processos com dados. Objetos com relacional.

Primeiramente a separação de responsabilidades no Modelo de Domínio são ainda maiores do que nos padrões exemplificados anteriormente, e se no Modelo de Tabela é recomendado utilizar uma ferramenta de ORM, no Modelo de Domínio isso é praticamente obrigatório.

É sempre bom frisa, que um Modelo de Domínio só é recomendado quando a regra de negócio é complexa demais, caso a regra seja simples, o recomendado é um Roteiro de Transação. Agora devemos criar uma classe que represente produto, como mostra a **Listagem 3**.

Listagem 3. Classe TProduto do Modelo de Domínio

```
TProduto = class
  private
    prv_sNome : string;
    prv_dValor : Double;
  public
    property Nome: string read prv_sNome write prv_sNome;
    property Valor : Double read prv_dValor write prv_dValor;
    function fTemNoEstoque : Boolean;
end;
```


DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DevMedia

Como foi dito anteriormente, que um Modelo de Domínio ele possui tanto dados como processos, ou seja, funcionalidade. A classe TProduto ela tanto tem as informações sobre o produto quanto o estado do mesmo, ou seja, permite que se verifique o estoque.

Agora vamos criar uma nova classe chamada TItensVenda, que conterá todos os produtos necessário para a venda e é agora que teremos as informações referente ao valor total dos produtos. Veja na **Listagem 4** a sua criação.

Listagem 4. Classe TItensVenda

```
TItensVenda = class
  private
   prv_oProdutos : TObjectList<TProduto>;
  public
   function fValorTotalCompra : Double;
   procedure pAddProduto(pProduto : TProduto);

   constructor Create;
   destructor Destroy; override;
end;
```

Temos um objeto do tipo TObjectList<TProduto>, que é a lista de produtos que serão comprados pelo cliente e esta variável está sendo instanciada no construtor e consequentemente destruída no destrutor. O Método pAddProduto dispensa grandes comentários, pois ele faz exatamente o que o seu nome diz que faz, que é adicionar o produto na lista. Já a função fValorTotalCompra faz um lanço para saber o valor total da compra como mostra a **Listagem 5**.

A implementação é bastante simples, que até despensa comentários. Agora para concluir devemos criar uma última classe como mostra a **Listagem 6**, que efetivará a venda.

Listagem 5. Implementação da função fValorTotalCompra

```
function TItensVenda.fValorTotalCompra: Double;
var oProduto : TProduto;
begin
 Result := 0;
 for oProduto  in prv_oProdutos.ToArray do
  Result := Result + oProduto.Valor;
end;
```

Listagem 6. Implementação da classe TVenda

```
TVenda = class
  private
   prv_oItensVenda : TItensVenda;
  public
   property ItensVenda: TItensVenda read prv_oItensVenda write
   prv_oItensVenda;
   function fFinalizarVenda : Boolean;
end;
```

Onde o método fFinalizarVenda conterá todo o código necessário para finalizar a venda e retornar se foi bem sucedida ou não.

A grande queixa de se utilizar o Modelo de Domínio é porque ele utiliza categoricamente os conceitos de orientação a objetos e para aqueles que não têm familiaridade com a orientação a objetos terão grandes dificuldades para entender e para aplicar na prática esse modelo.

Para implementar esse padrão levará bem mais tempo que se levaria para implementar um Roteiro de Transação e um Modelo de Tabela, mas os benefícios que o mesmo traz futuramente para o projeto é imensurável. Vale a pena utilizar esse modelo apesar de sua complexidade.

Conclusão

Agora a par dos Padrões de Arquiteturas de Aplicações Corporativas, já podemos implementar nos nossos projetos futuros, mas sempre tomando bastante cuidado para escolher o padrão certo para o projeto certo, pois uma escolha errada poderá trazer grandes estragos para o projeto.

Caso escolha um Roteiro de Transação e perceba que deveria ter escolhido um Modelo de tabela ou um Modelo de Domínio, poderá resolver esse problema com uma refatoração, que dependendo da complexidade do código em que se quer migrar, poderá ser fácil ou até um pouco mais difícil, mas caso escolha um Modelo de Domínio e perceba que deveria ter escolhido um Roteiro de Transação essa mudança é praticamente impossível.

Vale ressaltar que o foco desse artigo não era de lhe ensinar como fazer a regra de negócio, como persistir as informações no banco de dados, mas de mostrar como montar uma arquitetura capaz de durar longos ciclos de vida e que seja fácil a sua manutenção.

Welson Play
welsonplayqs@gmail.com
Programador Delphi há quase dois anos, possui a certificação Delphi Developer. Trabalha atualmente no desenvolvimento de sistemas em três camadas utilizando a tecnologia DataSnap e na criação e manutenção de framework de ORM.

Site do Martin Fowler
http://martinfowler.com/

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

CURSOS ONLINE

.net magazine

A Revista .net Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

MAIS de 20 CURSOS DISPONÍVEIS

CONHEÇA OS CURSOS MAIS RECENTES:

- Curso Básico de C#
- C# 5 e suas novidades
- Integrando Aplicativos ASP>NET C# com redes sociais (Twitter e Facebook)
- Refatoração com C# .NET
- ASP.NET MVC – Sistema de Vestibular

Para mais informações :

www.devmedia.com.br/curso/netmagazine

(21) 3382-5038

http://www.devmedia.com.br

DevMedia

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486